RL-TR-93-1
Final Technical Report
March 1993

# HIGH-PERFORMANCE DISTRIBUTED SYSTEMS ARCHITECTURE

BOLT BERANEK AND NEWMAN INC.

Carl D. Howe, Kenneth J. Schroder, A.D. Owen,
Richard Koolish, Charles Lynn

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-93-1 has been reviewed and is approved for publication.

APPROVED:

THOMAS F. LAWRENCE
Project Engineer

FOR THE COMMANDER

John A Graniero

JOHN A. GRANIERO
Chief Scientist for C3

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL ( C3AB ) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

| | | |
|---|---|---|
| 1. AGENCY USE ONLY (Leave Blank) | 2. REPORT DATE<br>March 1993 | 3. REPORT TYPE AND DATES COVERED<br>Final Aug 90 - Feb 92 |

4. TITLE AND SUBTITLE

HIGH-PERFORMANCE DISTRIBUTED SYSTEMS ARCHITECTURE

5. FUNDING NUMBERS

C - F30602-90-C-0036
PE - 62702F
PR - 5581
TA - 21
WU - 95

6. AUTHOR(S)

Carl D. Howe, Kenneth J. Schroder, A. D. Owen,
Richard Koolish, Charles Lynn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Bolt Beranek and Newman Inc.
Systems and Technologies Division
10 Moulton Street
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

7711

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory (C3AB)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-93-1

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Thomas F. Lawrence/C3AB/(315) 330-2805

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The objective of the High Performance Distributed Systems Architecture (HPDSA) project are to integrate very high bandwidth networks with heterogeneous computer architectures (including parallel and specialized processors) and support multiple programming models with this system. The driving forces for this project are user demands for high-performance computing coupled with the availability of gigabit and high bandwidth networking. Command and control applications are increasingly requiring the capabilities of both specialized processors and supercomputer-class machines. These resources are often concentrated at computer centers while the data sources and users of these systems are geographically distributed. By linking users, data sources, and systems together with high-speed networks and distributed operating systems, we can substantially improve the quality of command and control information and provide that information to more people. A high-performance distributed operating system must satisfy a number of design goals to accomplish our architectural objectives. Foremost, it must be capable of accommodating a wide variety of computers, networks and programming languages.

14. SUBJECT TERMS

High Performance Distributed Systems, Gigabit Network,
Distributed Operating System, Programming Models

15 NUMBER OF PAGES

52

16 PRICE CODE

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT<br>UNCLASSIFIED | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>UNCLASSIFIED | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>UNCLASSIFIED | 20. LIMITATION OF ABSTRACT<br>UL |

# Table of Contents

# Table of Figures

# 1. Introduction

The objectives of the High Performance Distributed Systems Architecture (HPDSA) project are to

- integrate very high bandwidth networks with
- heterogeneous computer architectures (including parallel and specialized processors) and
- support multiple programming models with this system.

The driving forces for this project are user demands for high-performance computing coupled with the availability of gigabit and high bandwidth networking. Command and control applications are increasingly requiring the capabilities of both specialized processors and supercomputer-class machines. These resources are often concentrated at computer centers while the data sources and users of these systems are geographically distributed. By linking users, data sources, and systems together with high-speed networks and distributed operating systems, we can substantially improve the quality of command and control information and provide that information to more people.

A high-performance distributed operating system must satisfy a number of design goals to accomplish our architectural objectives. Foremost, it must be capable of accommodating a wide variety of computers, networks, and programming languages. The performance demands and diversity of missions in $C^2$ environments require a heterogeneous computing and communications environment. This heterogeneous environment consists not only of general-purpose computers, but also supercomputers, parallel processors, graphics renderers, digital signal processors, and other specialized platforms. Each platform provides specialized resources to the computing environment, but the applications using these distributed resources require that they operate as a unified and consistent whole. A distributed architecture that provides a uniform programming model over a wide range of computing platforms and applications can achieve this goal.

The object-oriented programming model provides a framework for such a system. It allows users to have a consistent view of all services, while insulating the user from the implementation details of those services. This enforcement of modularity and the separation of policy and mechanism make the object model a powerful way to build large-scale, distributed applications. Therefore, support for the object model is an important goal in the HPDSA.

Nonetheless, we recognize that there are many applications that do not naturally fit into this model. This class of applications includes equation solvers, text searching algorithms, and demand-driven shared-memory systems. Therefore, our goal of supporting an object-oriented system must permit high-performance communication in both object-oriented and non-object-oriented environments.

We also recognize that our architecture must be scalable to accommodate large user and service populations. As military networking expands to provide near-universal connectivity, the range of services that users of the network require will also expand. In particular, we expect that there will be increasing information flow across service boundaries to support $C^2$ operations, and it is critical that the HPDSA support and not hinder that expansion. Therefore, the facilities incorporated in the architecture are designed for efficient large-scale operation (e.g., thousands to millions of hosts) to accommodate this expansion.

Finally, we accept as a goal that the HPDSA must be as simple and easy to understand as possible. While high-performance operation is a requirement of this effort, it should not come at the cost of increased complexity for the programmer or the user. The programmer should be able to use natural abstractions in applications and to apply criteria to those abstractions to achieve high performance.

During the course of this project, we have designed a distributed system called Photon that attempts to satisfy the above goals. The design is optimized to work in environments where the ratio of communications latency to computational power and bandwidth is relatively high compared to what it is today. In other words, it anticipates improvements in communications bandwidth and computational power, but assumes speed of light will not improve.[1] This assumption implies that the architecture of the system must be able to accommodate greater parallelism and less direct feedback from remote locations to achieve high performance.

Photon has incorporated a few key mechanisms to accommodate high-latency environments. One of these is globally-claimable futures (also referred to as global futures). *Futures* [Halstead Multilisp] define points in a program where parallelism can be used while simultaneously specifying a result that will force synchronization between the caller and the called function. Globally-claimable futures allow this abstracted result to be passed to other computers on the network, where they can be claimed by other programs requiring the data. These globally claimable futures permit the data to flow directly to the programs where the results are used rather than conforming to an artificial client/server distinction. This fact eliminates needless "round-trips" and thereby expedites program execution in high-latency environments.

Globally-claimable futures allow programs to explicitly declare their data dependencies and to implicitly define available parallelism. However, without proper integration into the programming environment, the concept of globally-claimable futures would be difficult to

[1] Should the speed of light increase at all, the system would still function, but would probably benefit from redesign along with many of the physical sciences.

use and manage. To achieve this integration, we borrow a concept previously used in the Argus system [Liskov Argus] called *language veneers*. Language veneers permit the basic functions of a distributed system to be incorporated into a high-level language through small syntactic additions to the language. These additions are tailored to the style and flavor of the language to provide the most natural interfaces to the functions provided by the underlying system. In our case, we concentrated on defining a language veneer for the language C++, which allows us to mate an object-oriented programming language with an underlying object-oriented distributed system. This veneer permits C++ objects and operations to map onto corresponding distributed objects and operations. Our veneer permits the programmer to use a very natural and sequential programming style while actually invoking futures and data-flow mechanisms. The result of this is that distributed programming in Photon can resemble natural C++ programming.

To prove that these mechanisms can promote high performance distributed applications, we wrote a series of demonstration programs using prototype implementations of these mechanisms. These ranged from simple echo programs to parallel computations of the Mandelbrot set. The language veneer was demonstrated by writing a set of class definitions that allowed construction of very simple distributed programs through normal C++ object definitions and references. While each of these demonstrations was hand-coded to demonstrate a specific aspect of Photon, all of these demonstrations make use of library functions that implement the basic Photon concepts rather than an external "kernel". By embodying Photon mechanisms within the user program's address space, context-switching time within a computing platform is eliminated, and no one process becomes a bottleneck when multiple processors are used in parallel.

These demonstrations are a first step in the development of a distributed programming environment capable of exploiting gigabit networks, parallel processors, and multiple programming models. We believe that they introduce new concepts to work around fundamental physical limits to the speed of distributed applications. As processor and networking technologies evolve through the end of the twentieth century, we believe that concepts like these will play an increasingly important role in defining distributed applications.

## 1.1. Document Overview

This document describes the environment, issues, and mechanisms involved in the design of Photon, our high performance distributed systems architecture. While it attempts to provide a broad overview of the architecture and the problems it attempts to solve, it does not present all elements of the design in depth. To avoid unnecessary duplication, we refer readers wishing to understand the design in greater technical detail to the following project documents:

- Technical Report #7707, System/Subsystem Specification for Photon, A High-Performance Distributed Systems Architecture, Jan. 1992.
- Technical Report #7708, Software Design Document for Photon, A High-Performance Distributed System Architecture, Feb. 1992.
- Technical Report #7709, Photon Software User's Manual, Feb. 1992.

This document builds upon research reported upon in our Technical Report #7619, High-Performance Distributed Systems Architecture Interim Technical Information Report released June 1991. That report described the issues and background that we believed to be the driving forces behind this project. We have included portions of that prior work in this report for the sake of completeness and convenience to the reader. This report supersedes the prior interim report and attempts to summarize the entire project effort.

## 2. High-Performance Computing and Network Environment

The computing environment of the 1990s is the key driving force for the HPDSA. Advances in computer architecture, computer networking, and systems integration lead one to believe that existing distributed computing techniques will be scalable to an environment consisting of gigabit networks, teraflop mainframes, and ubiquitous access by users. While these techniques may work, there are constraints imposed by physics that will limit the performance of traditional techniques, and will require us to consider significant changes in distributed architectures.

Certainly the most visible advances in computer systems technology have been in microprocessor speed. We can get a sense of this advance by examining the speeds of single-processor workstations and using them as indicators for computing in general. In 1985, the 68020 had just been introduced, and this was the first 32-bit microprocessor system actually delivering more than 1 MIPS in an inexpensive workstation (Sun 3/50). Today, the HP Apollo series 700, the IBM RS 6000, and the Sun SPARCstation 2 all deliver in excess of 20 MIPS and some as high as 76 MIPS. The outstanding question is, can the microprocessor designers and semiconductor houses continue this improvement over the next five years?

All evidence currently indicates that not only can they continue to improve at this rate, but the rate of improvement could accelerate. The largest contributing factor to this acceleration is Reduced Instruction Set Computer (RISC) technology. RISC technology has two simplifying characteristics: the basic functions of the system are small and simple, and hard problems to solve in silicon can be pushed off into software, making the chips simpler yet. Simpler chips make it possible to produce working systems faster, and this, in turn, results in faster systems sooner.

Figure 1 shows the basic processor speeds for each year beginning in 1985, and a rough projection of this trend through 1995. Several representative processor families, including the Motorola 680x0 and 880x0, and Intel 80x86, are shown in this graph, along with projections made by Prof. John Hennessy of MIPS Computer Systems and R. Andrew Heller of the IBM workstation division [Microprocessor Report]. An analysis of this data shows that we have been seeing performance increases of approximately a factor of 1.85 per year for the past five years, and this appears to be a conservative factor for the next five. This implies that we will have chips in 1995 that compute more than 450 MIPS. Equally impressive is the prediction of more than 130 MIPS in 1993. Comparing these estimates with what semiconductor companies believe they are designing, these numbers are quite reasonable and can be considered a conservative estimate.[2]

It is important to ask ourselves if we believe this to be reasonable growth. Certainly, there is no natural law that guarantees the computing community exponential increases in speed. In fact, any technology curve actually is more like an S curve, as shown in Figure 2. Therefore, we have reason to be skeptical of continued exponential growth. Presently, however, there do not appear to be any significant obstacles to achieving the performance shown in Figure 1. The technology used involves shrinking the minimum line size of the chips, reducing the voltages used on the chips to increase clock rate, and executing multiple instructions per cycle to achieve fine-grained parallelism. While these achievements are significant, they are fairly straightforward from an engineering point of view, and they can leverage off existing designs in supercomputers. Furthermore, we have extrapolated only the performance numbers for CMOS integrated processors. If we permit the use of more exotic processes such as ECL and gallium arsenide, we can even increase these estimates. However, CMOS will continue to be the mainstream technology for the foreseeable future, and therefore it provides a convenient lower bound.

Perhaps the most significant issue not shown in Figure 1 is that of word size. At present, one of the differentiating factors between supercomputers and mainframes is that most supercomputers are considered "true" 64-bit machines, whereas mainframes still use 32 bits for almost all integer and addressing operations. However, as companies like Digital Equipment Corporation and MIPS Inc. introduce chips such as the Alpha and M4000 respectively, this distinction will begin to vanish. In all likelihood, this change will be as significant as the jump from 16 bits to 32 bits was in the 1970s, and it will dramatically affect the way programs are written and run, just as 32-bit processors did in the 1980s.

At present, we expect the jump to 64 bits in workstation class machines to occur sometime between 1992 and 1993. Not many systems other than parallel processors are feeling a need for 64-bit address spaces at present. Nonetheless, based upon technology announcements such as the DEC Alpha architecture, we can expect all competitive workstation microprocessors to be 64-bit by 1995, and that no significant new systems except for personal computers will be designed around 32-bit processors beginning then. Because of the installed base of 32-bit processors, this change implies that throughout the 1990s, distributed systems will have to

[2] Actually, these numbers already appear excessively conservative. BBN's estimate was for 40 MIPS in 1991, and HP has nearly doubled that with its series 700 model 750 at 76 MIPS. Intel's David House has publicly stated that the Intel P5/80586 architecture will offer performance in the 100 MIPS range in 1992 [EE Times]. Digital Equipment has announced their Alpha chip should result in workstations running at 150 Specmarks in 1992 and achieving 400 MIPS soon after.

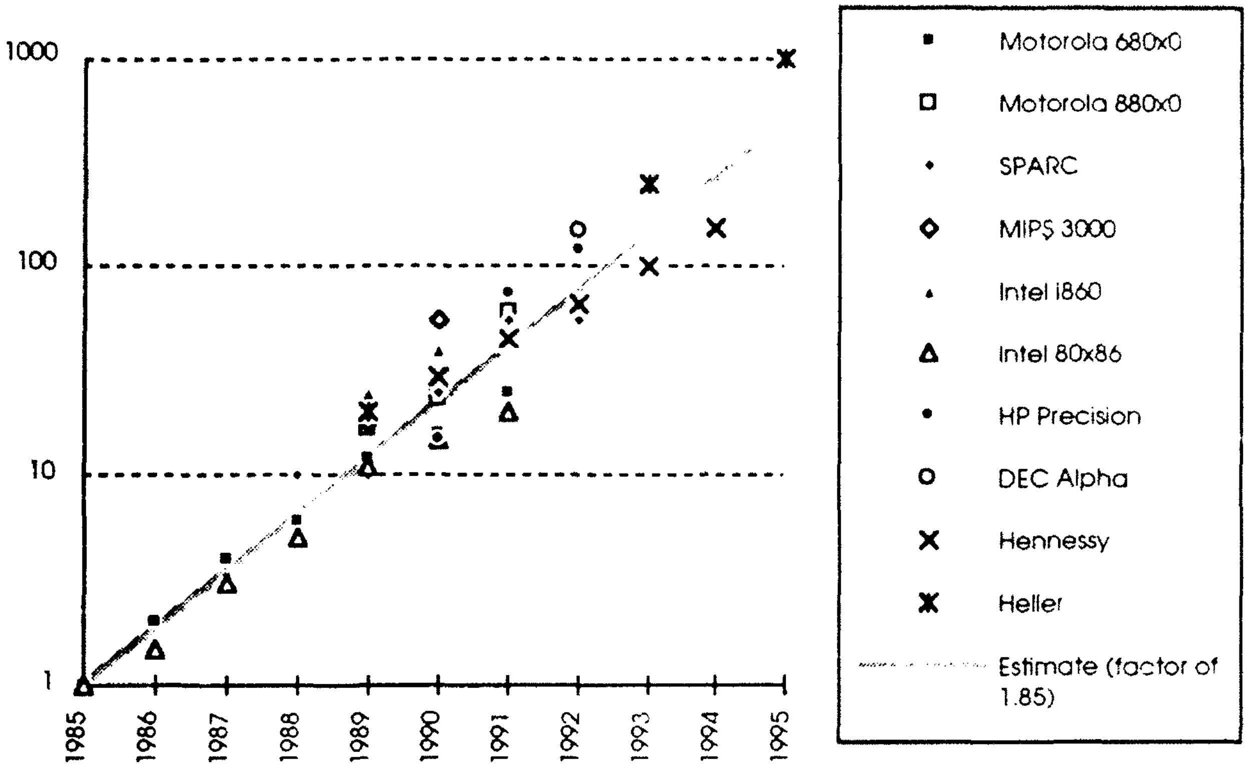


**Figure 1: Projected Microprocessor Performance in MIPS**

support both 32-bit and 64-bit architectures. It also implies that distributed systems will become increasingly heterogeneous with regard to word size as well as instruction set.

Compared with computer systems, communications technology has been relatively simple to evaluate until the last few years. Communications bandwidth cost and performance were inversely proportional to the distance over which the communication occurred. This resulted in local area networks having the highest bandwidths and the lowest cost per bit transmitted, and wide-area networks having the lowest bandwidths and the highest cost per bit transmitted.

This relationship can be seen easily by comparing Ethernet, as an example of current local area network (LAN) technology, with the Terrestrial Wideband Network, an example of wide area network (WAN) technology. Ethernet provides a peak bandwidth of around 10 Mbits/sec and can provide communications over a distance up to around 1 mile. Worst-case transmission latency is on the order of tens of microseconds to any point on the Ethernet. The TWBNet, on the other hand, provides peak bandwidths of 1.44 Mbit/sec, but can provide national and international communication. However, worst-case transmission latency is also worse, reaching values of 250 milliseconds when satellite links are involved.

Because of the diverse characteristics of WAN and LAN technologies, the most successful large-scale communications strategies have been combinations of the two. The premiere example of such a combination is the Defense Research Internet. In the Internet, local communication usually takes place over a LAN. Since the vast majority of data sent on a network tends to be addressed to local users, the high bandwidth of the LAN is used to provide low-cost transmission whenever possible. Data to be transmitted to areas not served by the LAN is sent by a gateway onto one or more intermediary networks. The data only travels as high in the hierarchy of networks as is necessary for it to be transmitted to its destination. This permits expensive, long-haul networks to be used only when long-haul communication is required. The Internet thereby makes efficient use of the pricing and bandwidth characteristics of both LANs and WANs for achieving global communications.

In the 1990s, this formula relating bandwidth, coverage, and communications cost is likely to become obsolete and thereby change our ways of thinking about communications networks, at least within the continental United States. The technology driving this change is the deployment of large amounts of fiber-optic cable by the

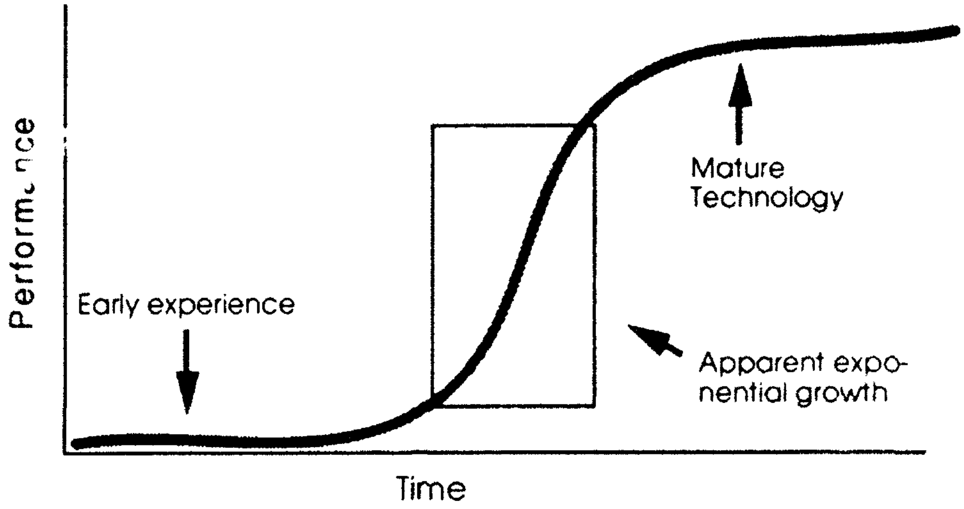


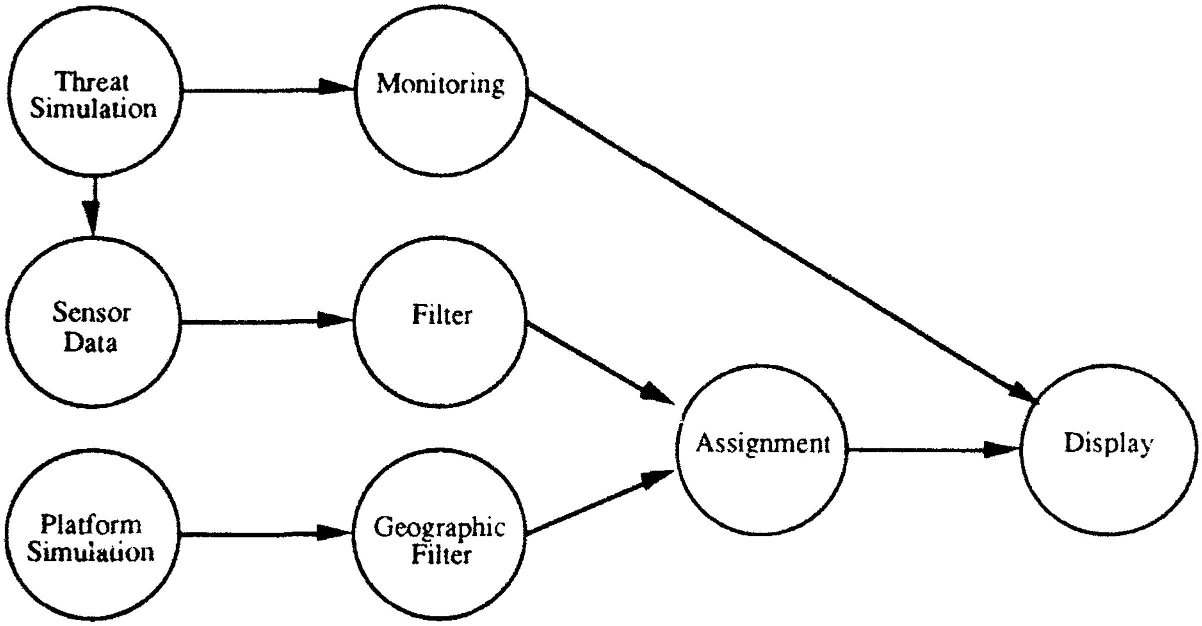
**Figure 2: Generalized Technology Curve**

regional, national, and international phone companies. To date, most of this deployment has taken place between switching offices of the telephone companies, and therefore has been largely invisible to the communications consumer. However, the technology is now becoming visible in the form of new services such as SONET and BISDN, which are available to almost any subscriber with the need for high-bandwidth communication. National and world-wide packet networks such as the NREN and DARTNET are providing peak access bandwidths of between 1 and 45 Mbits/sec. today and will reach gigabit speeds within the decade.

One aspect of WAN communications that will not improve as significantly as it has in the past is latency, the time it takes for a message to travel from one point to another. In the past, network vendors have been able to improve latency by using higher bit rates for network host interfaces and techniques such as *cut through* [3] in packet switches. However, no matter what improvements in technology occur, network latency will be fundamentally constrained by the time it takes light to travel between those two points. While use of fiber-optic technology may permit sending bits at a rate in excess of 1 Gigabit/sec from Boston to Los Angeles, those bits cannot arrive in Los Angeles any sooner than 14 milliseconds after they are sent (assuming a 2500 mile, straight-line, fiber link between Boston and Los Angeles). Furthermore, we cannot expect to receive an answer from Los Angeles any sooner than 28 milliseconds after we send the question from Boston. A way to think about this constraint is to imagine interstate highways between Boston and California that have 4000 lanes, but with a strictly enforced 55-mile-per-hour speed limit. While these roads can handle significantly more cars than ones with four lanes, this increase in capacity does not change the time it takes to get from one coast to the other.

While these best-case latency numbers are very good, they are within an order of magnitude of the latencies we are experiencing on wide area networks today. Therefore, while wide area bandwidths are going to improve by almost three orders of magnitude (1.44 Mbits/sec to 1 Gbit/sec), cross-country latencies can at most improve one order of magnitude (250 milliseconds to 14 milliseconds, one way). Therefore, the ways we use our networks will have to change qualitatively to achieve performance improvements proportional to the increase in bandwidth. Distributed systems in particular will have to recognize the latency constraints inherent in WANs and incorporate facilities to mitigate their effects on programmers and applications.

[3] Cut through is a technique where a packet-switch can begin transmitting a received packet as soon as it has read its header and knows where it should direct it. In essence, this allows the packet to begin transmission along a new link before it has been fully received on a previous one, thereby reducing latency due to the intervening packet switch.



| Static Mechanisms | | Adaptive Mechanisms | |
|---|---|---|---|
| Specialized Modules | ✓ | Many Identical Modules | |
| Centralized Control | | Decentralized Control | ✓ |
| Static Load Balancing | ✓ | Dynamic Load Balancing | |
| Point-to-Point Communication | ✓ | Multicast Communication | |
| Reliable Communication | ✓ | Unreliable Communication | |
| Requires System Fault Tolerance | ✓ | Fault Tolerance in Application | |

**Figure 3: Technology Validation Experiment Application Schema and Application Characteristics**

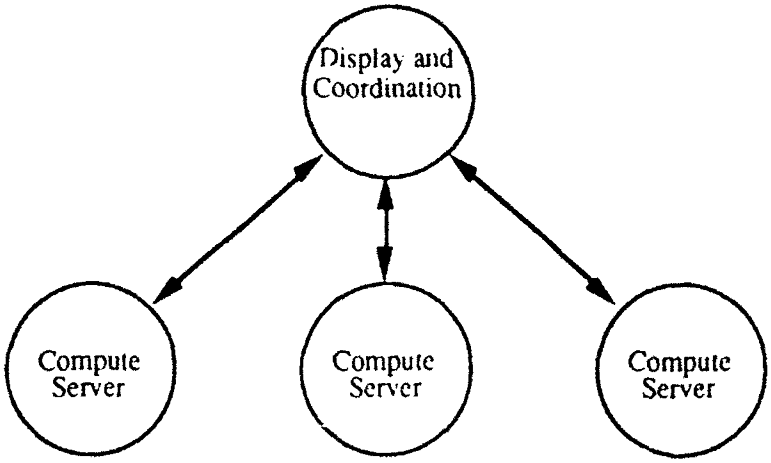
# 3. Discussion

## 3.1. Example Applications

We chose to evaluate a small group of distributed applications before we started our HPDSA design. We believe these applications to be typical of classes of problems solved by distributed systems. We did this to provide a firm technical foundation for design decisions and to ground our architecture in the needs of real world systems. By determining the architectural and performance needs of these applications, we can extrapolate those needs into the a list of requirements for our HPDSA. Furthermore, these applications provide us with a basis for evaluating the features of our design after it is complete.

### 3.1.1. Technology Validation Experiment (TVE)

The first application we considered was the Technology Validation Experiment or TVE [Schroder TVE1][Schroder TVE2]. This application was based upon a collection of programs written by MITRE to simulate the detection, tracking, and weapon engagement of missile threats during their boost phase. The distributed version of the system was created using Cronus [Schantz Cronus] to validate its ability to incorporate existing code in a distributed environment.



| Static Mechanisms | | Adaptive Mechanisms | |
|---|---|---|---|
| Specialized Modules | | Many Identical Modules | ✓ |
| Centralized Control | ✓ | Decentralized Control | |
| Static Load Balancing | | Dynamic Load Balancing | ✓ |
| Point-to-Point Communication | ✓ | Multicast Communication | |
| Reliable Communication | ✓ | Unreliable Communication | |
| Requires System Fault Tolerance | | Fault Tolerance in Application | ✓ |

**Figure 4: Mandelbrot Set Computation Schema and Application Characteristics**

The overall TVE simulation system consists of three major sections: threat generation components, simulation components, and a display subsystem for monitoring the results of the experiment. The threat generation computes trajectories and orbital propagation for a collection of simulated attacking missiles. The simulation components then generate sensor data based upon these trajectories, filter that data, evaluate feasibility of intercept by orbiting weapon assets, and perform weapon assignments. The display system presents the threats and the results of the simulation to the user of the system. A simplified diagram of the application and the application's attributes are shown in Figure 3.

The TVE exhibits several interesting characteristics with regard to the underlying distributed architecture. In essence, this application has been decomposed by function, and each function has been placed in a separate module. The distributed system provides communication and synchronization among those modules, but this is largely limited to data translation and flow control. The communication is simple, point-to-point, and one-way; and no significant use is made of objects below the module level.

In many ways, these characteristics are typical of a straightforward data-flow application. Each specialized module waits until it has the data it needs. Once a module's inputs have been supplied, it performs a computation, sends the output of that computation on to the next module, and waits for more input. All synchronization and flow control is provided by the distributed system's communications mechanism, and the longest path through the system sets an upper bound on how responsive the system can be.

### 3.1.2. Mandelbrot Set Computation

The second application we looked at was the computation of the Mandelbrot Set by a collection of computers. This computation is basically the classification of points in the complex plane into categories of convergence or divergence based upon successive evaluation of a function. This classification of the entire complex plane is computationally expensive, but this computation can be run in parallel easily because the classification of each point is independent of that of all other points in the plane.

This example is representative of parallel processing applications programmed using a *task bag* model (see Figure 4). The basic concept is that there are one or more processes that create descriptions of work to do, and those descriptions are then placed in a heap or *task bag*. At the same time there are several compute server processes that request work descriptions from the task bag, perform the computation described, and return a result. The results are then collected and displayed, usually by the process that initiated the computation.

One interesting aspect of this application is that it uses multiple computation servers that are identical in function. The number of computation servers available is not significant to the application; it will run properly with one computation server or one hundred. By structuring the application to use this task bag model, the object code becomes independent of the number of processors used to run it.

Furthermore, provided the number of points to be classified is large, the computational load is automatically balanced across all these processors regardless of their relative speeds. Slower processors will request new work relatively slowly while fast processors will request work more frequently. This behavior maximizes use of all computation servers available to the computation and provides maximum throughput.

For extremely large computations such as factoring many-digit composite numbers, the user may want to use hundreds or thousands of computation servers. This need introduces a requirement that the computational node with the task bag be able to communicate with hundreds or thousands of other machines simultaneously. Alternatively, the task queue may have to be distributed across several processes to avoid communication and processing bottlenecks.

### 3.1.3. Pilot's Associate

This application is a theoretical collection of programs that assist a pilot in threat identification and weapons targeting. The model used is one where many sensors provide information regarding possible threats around an aircraft. As information regarding these possible threats arrives, the information is posted in a common data structure. Multiple threat assessment processes examine the information posted in this database asynchronously to determine which threats are most dangerous to the aircraft, and these threats are identified to the pilot for action. The schema for this application is shown in Figure 5.

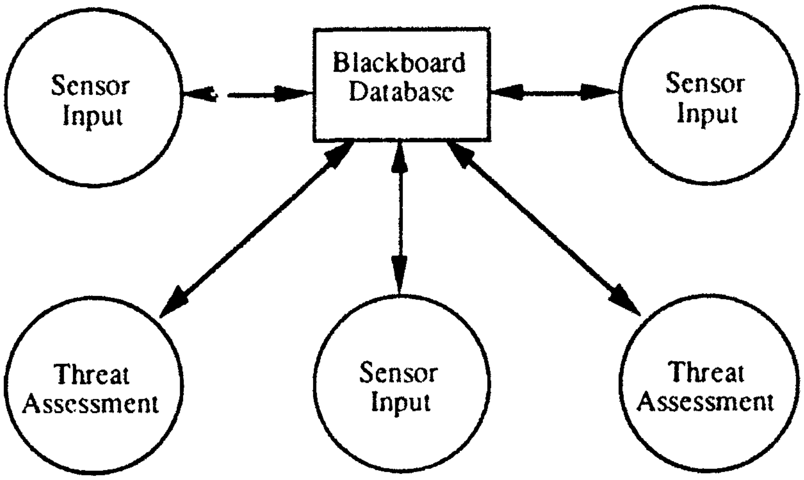
In many ways, this application is a synthesis of the first two. Data is still flowing from sensor processes to threat assessors, but this flow is mediated by a common data structure or *blackboard*. Multiple threat assessment computations occur simultaneously, and these provide dynamic load balancing. Finally, the application can make use of any number of sensor and threat assessment processes without modification and has some inherent fault-tolerance. Should a threat assessor or sensor fail, only data associated with that module with be lost, and all other computations will be able to continue.

Of particular interest in this application is the fact that the input and assessment processes are only loosely coupled. The sensor processes store data in the blackboard as it becomes available, without regard for whether the previous data has been read yet or not. Similarly, each threat assessment process acts upon the available threat information to determine the best target at this moment, without regard for explicit synchronization. Although it is drastically simplified, this model is typical of real-time systems that must be able to function in spite of possible "input overload" scenarios. Therefore, we must ensure that our system be able to support loosely-coupled as well as explicit synchronization techniques.

While it may not be readily apparent, this application is representative of a larger group of systems such as shared workspace or conferencing applications. We mention these systems because they have broad application within military command and control environments. They differ because people represent the sensors and threat assessors in conferencing and shared workspace systems while the pilot's associate application uses automated subsystems in those areas. Nonetheless, the issues of noncentralized control, loose synchronization, and shared workspace are common between the two application areas. We believe that by addressing the issues raised by the pilot's associate, the HPDSA should be capable of supporting a wide range of blackboard and shared-workspace systems, including electronic conferencing.

### 3.1.4. Advanced Simulatl^n

Our fourth distributed application is based upon battlefield simulations such as those implemented in the DARPA Advanced Simulation Program. In this type of simulation, each vehicle on the battlefield is simulated by a separate computer on the network. Each vehicle simulator broadcasts a packet describing its position and appearance to all other simulators periodically. Every vehicle simulator is responsible for modeling its behavior and its interactions with other vehicles by listening to all the broadcast packets from other simulators. If a simulator does not hear a new position and appearance packet from another simulator it is interacting with, it extrapolates the remote machine's appearance and position from the last data it had; therefore, the system tolerates unreliable communication. Furthermore, to enhance realism, should a simulator fail during a simulation, another simulator will repeat its state and appearance pack-



| Static Mechanisms | | Adaptive Mechanisms | |
|---|---|---|---|
| Specialized Modules | | Many Identical Modules | ✓ |
| Centralized Control | | Decentralized Control | ✓ |
| Static Load Balancing | | Dynamic Load Balancing | ✓ |
| Point-to-Point Communication | ✓ | Multicast Communication | |
| Reliable Communication | ✓ | Unreliable Communication | |
| Requires System Fault Tolerance | | Fault Tolerance in Application | ✓ |

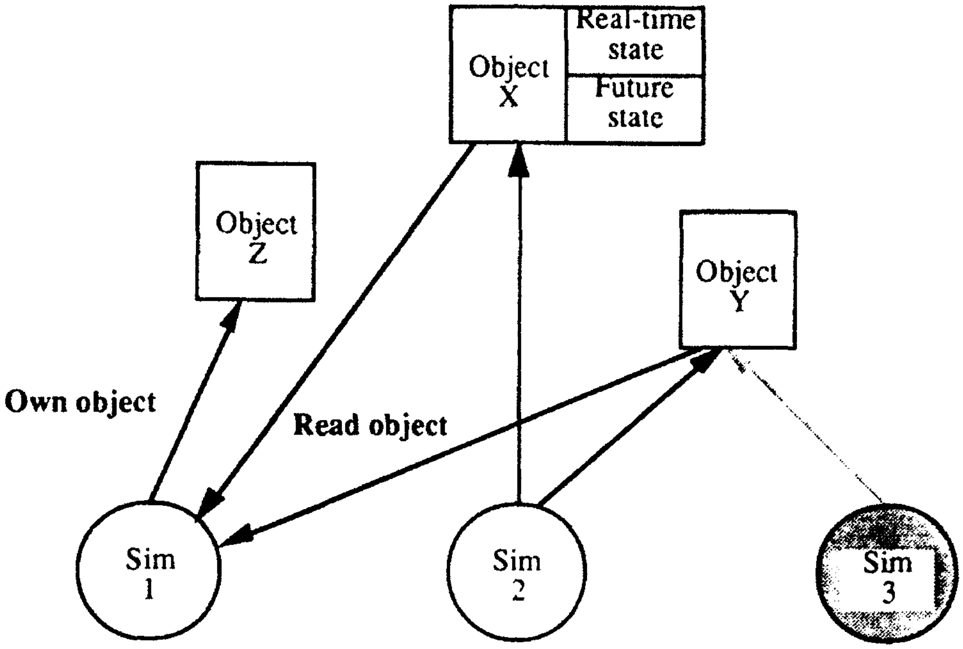
**Figure 5: Pilot's Associate Computation Schema**

ets until the simulator comes back on line. A computational schema is shown in Figure 6.

This simulation application is an object-oriented environment where the objects correspond one-to-one with the computational nodes of the network. It is similar to the blackboard-based pilot's associate application because of the loose coupling between the various simulators. However, unlike the pilot's associate, the central blackboard is distributed throughout the network. All simulators maintain their own view of the state of the world and update it according to broadcast packets from other simulators.

This application also differs from the previous applications because it attempts to provide object persistence even when the simulator representing a specific object fails. Therefore, while objects are usually mapped one-to-one to simulators, that mapping can change as the result of failures.

One final aspect that is significantly different in this application is its use of communication. This system requires the use of timely broadcast or multicast communication. While it can tolerate unreliable distribution, the structure of the system demands that every simulator hear most packets sent by all other simulators and that all packets be delivered within a fixed time period referred to as a simulation *frame*. This requirement differentiates the system from many other applications and requires support for this timely delivery from the underlying distributed operating system.



| Static Mechanisms | | Adaptive Mechanisms | |
|---|---|---|---|
| Specialized Modules | | Many Identical Modules | ✓ |
| Centralized Control | | Decentralized Control | ✓ |
| Static Load Balancing | ✓ | Dynamic Load Balancing | |
| Point-to-Point Communication | | Multicast Communication | ✓ |
| Reliable Communication | | Unreliable Communication | ✓ |
| Requires System Fault Tolerance | | Fault Tolerance in Application | ✓ |

**Figure 6: Advanced Simulation Computation Schema And Application Characteristics With Failure of Simulator 3**

# 4. Issues

Based upon our studies of these applications, we constructed a list of issues that we believe must be addressed by the HPDSA. These issues are presented in the forms of questions that features of the architecture should address. These issues are grouped into the following areas.

- Communications
- Computation
- Binding
- Naming
- Fault-tolerance and correctness
- Resource Allocation

Each area will be discussed separately in the following sections.

## 4.1. Communications

Our sample applications demonstrated a wide range of communications needs. Next-generation, high-bandwidth networks will also dramatically change the character of the infrastructure that must service those needs. Therefore, our architecture should address the following communications issues.

*How do we deal with proportionally greater communications latency?* As we described in section 2, the next-generation of computer networks will have proportionally greater latency than networks of today. In other words, their latency will not improve nearly as dramatically as their bandwidth will. Our new architecture must recognize this qualitative change in network characteristics and provide mechanisms for mitigating its effects.

*How do we model scheduled versus unscheduled communications (e.g., connection-based models versus datagram models)?* To date, most distributed applications have been designed for specific networking substrates, and have not tried to negotiate their communication needs with the communications system. Next-generation applications will make specific demands of the communications infrastructure, such as low delay, best-effort delivery, and reliable delivery and sequencing. Some of these demands will be known in advance (i.e., we'll know the needs are there when we design the application), while others may not be known until the application is run. We require facilities within the HPDSA for the application to declare its communications needs. The distributed system will then map those needs onto the available communications protocols and interfaces.

*What choice of communication semantics must be provided (e.g., reliable delivery, rapid delivery, sequenced delivery)?* Not all applications require reliable delivery, and some (such as advanced simulation or real-time videoconferencing) may be degraded by latency variations and retransmissions implied by this type of delivery. The distributed architecture must be able to accommodate the specific communications needs of applications without overloading them with unneeded communications features.

*How does the user specify the communications needs of the application?* Given that the architecture must be able to satisfy the application's needs, how does the application specify those needs to the system? Distributed applications are often insulated from the communications medium, and therefore have had little opportunity to control their communications environment.

*How do we provide communications to many objects at once?* In many systems, there are limits to the number of communication paths or virtual circuits that can be open simultaneously. This type of restriction should not be made visible to the applications programmer. The user should be able to work with as many or as few objects as are required by the application, and the distributed operating system should support that use without burdening the programmer with excessive communications bookkeeping.

In some communications environments, multicast facilities may be available. The use of multicast protocols can substantially reduce communications cost in these environments and can greatly improve the scalability of the system. However, because not all environments support multicast communication, the system must be able to make intelligent decisions regarding when multicast protocols can be used.

*How do communications abstractions interact with the object-oriented programming model?* Most object-oriented programming languages do not incorporate the concept of a communications model because all object interactions are contained within one process and are perfectly reliable. When we start locating objects at other nodes of the network, we must add concepts of communications errors and failure into the object model.

## 4.2. Computation

Because we must support parallel as well as distributed platforms in our architecture, we must clearly define a model of computation that includes parallelism. Arriving at this model of computation requires that we address the following issues.

*How do we express parallelism in a distributed application?* Compute-intensive distributed applications may improve their performance significantly by running parts of the computation in parallel. However, traditional sequential languages with synchronous Remote Procedure Call (RPC) models provide no opportunity to express

·his parallelism. The distributed system architecture must provide ways for the application to relax its sequencing and to express its parallelism to make maximum use of multiple processors in the distributed environment.

*How do we bind our parallelism abstractions to real computational elements to maximize the performance of the computation?* The specification of parallelism by the application should be based upon the structure and sequencing requirements of the computation. However, this expression of parallelism may result in significantly more computations that are able to run in parallel than we have processors to actually run them. For example, the computation of the product of two 100x100 matrices can be divided into 10,000 independent computations. However, it is not likely we'd have available (or want to pay for) 10,000 separate processors to perform this computation. Therefore, the distributed system must provide mechanisms for mapping abstract parallelism to real computational elements in ways that optimize performance, cost, or other parameters.

*How do we express data dependencies within a distributed application?* Once we have established methods of defining parallelism, we must ensure the correctness of the computation, regardless of the amount of parallelism being used. Traditionally, this is managed by the programmer by writing his or her program in serial fashion, with the assumption that every statement cannot proceed until the previous statement has finished. If we are going to be able to exploit arbitrary parallelism within a distributed system, the distributed system needs to know what dependencies are required for correct operation and what statements are independent from one another.

*How do our computational abstractions synchronize with one another?* Once we've provided parallelism abstractions within our distributed application, we must also provide ways for those computational abstractions to work together to provide consistent and deterministic results. In our matrix multiplication example, we would not want to print out the product matrix until we were confident that the computation of all 10,000 elements of the matrix had finished. Similarly, we would not want to update a record of a database at the same time that it was being written by an independent process. These synchronization methods must work in both distributed and parallel contexts and must be efficient enough that they do not significantly alter the overall performance of the application.

## 4.3. Binding

A distributed system must permit servers and clients to communicate. While these communications facilities might be completely unstructured, it is useful to maintain some sense of connection between the two entities, particularly to preserve ordering of operations, reliable transmission, and error detection and reporting. However, the maintenance of a connection raises the following issues.

*How do we create bindings between clients and servers?* If we wish to provide specific classes of service to an application (for example, we wish to be able to guarantee ordered, reliable delivery for all operations on a specific object), the distributed system must provide ways for the client and server to maintain state concerning their communication. This state information might include what operations are outstanding, which operations have completed, and what types of exceptions might be pending. In essence, this maintenance of state at each end of the communication path defines a communications connection or binding between the two entities. Since we desire that applications be able to specify performance and delivery requirements for their communications channels (as noted in the communications section above), the distributed system must have the ability to set up bindings between clients and servers as necessary to fulfill those requirements.

*At what time are these bindings created?* The existence of client-server bindings implies that they are created at some time and destroyed at another. While simple, static bindings might be acceptable for some applications, parallel applications (such as the Mandelbrot Set Computation) may require a more dynamic, unscheduled type of binding to perform load balancing. Furthermore, bindings must be dynamic enough to cope with server or network failures; should a server fail, the system may need to establish a new binding to a backup server.

*How long do these bindings last?* While bindings are useful when there are many operations being performed on a single object, they stop being useful when all operations on an object are completed. Furthermore, there is a cost to keeping unneeded bindings because they require storage of state in both the client and server. Therefore, the distributed system requires mechanisms for eliminating bindings once their usefulness is over.

## 4.4. Naming

*How do we name objects and services?* Distributed applications need ways to refer to objects and services with which they may not have communicated previously. A convenient way of permitting this reference is to attach names to those objects and to allow those names to be stored in messages and catalogs among the components of the system. The distributed system should provide a naming architecture for objects and to provide ways of deriving addresses for objects from their names.

*How do we map descriptive requirements (names, properties, keywords) to servers?* While hierarchical names are one method of locating objects, some applications may need to reference servers based on other criteria such as distance, available bandwidth to the server, administrative domain, and so on. While these properties can be expressed in a hierarchical name space, the concept of locating objects based upon properties is more akin to a database lookup by attribute than to a name lookup. Therefore, a distributed system may need to provide ways of locating servers and services other than just by name.

*What format and scope does an object name have?* Object names need to be manipulated by both people and computers. While variable-size, alphanumeric names such as `/my/favorite/files` may be convenient for people to remember, they are cumbersome to use if they are the only way of communicating with an object. At the same time, names that computers can manipulate easily (e.g., `0x35fc9987`) are not convenient for people to work with. We need to provide both ease of use and efficiency in our distributed system.

Similarly, we need to define the scope of these names. The scope of a name could be strictly local (e.g., only valid on the current machine), cluster local (only valid in the current administrative domain), or global (valid from any machine participating in the distributed system). Names with global scope could be absolute names (i.e., unique throughout the distributed system) or relative (qualified by a concept of current location within the name space). The HPDSA must resolve these issues to provide a coherent naming strategy for objects.

*Can an object name specify more than one object?* The name space provides a many-to-one mapping of alphanumeric strings to objects. However, a one-to-many mapping of alphanumeric strings to objects might also be useful. For example, one might wish to have a single name for a service such as `time_service` and have that one name be a reference to any of a number of time servers. Similarly, one might want to refer to a computation made up of many clients, servers, and objects by a single name for the purpose of starting, stopping, monitoring, and debugging the computation.

## 4.5. Programming Model Issues

*What is the model of computation offered to the programmer?* An integrated distributed system extends the programmer's model to include new collections of data and new procedures for manipulating that data. The Remote Procedure Call model provides the programmer with access to procedures located on remote machines, but does not permit direct access to the data located remotely. A virtual shared memory model provides direct access to remote data and does not require the programmer to use specific procedures for that access. However, implicit in this shared memory model is the assumption that the application understands the structure and meaning of that instance of shared data. Object-oriented programming models integrate both types of access by supplying procedures for operations upon objects and publicly defining the data types involved in those operations.

*How is this computation model integrated into the programming language?* The mechanisms defined by the underlying distributed system must be presented to the programmer as part of the language being used to build the application. Providing only one version of these mechanisms for all languages is difficult because not all programming languages work from a common set of assumptions. For example, a procedural interface to the distributed system's mechanisms would not be easy to use in a strictly functional programming language. Similarly, object-oriented interfaces are likely to be clumsy to use in non-object-oriented languages unless the interfaces are carefully designed to work in both environments. The HPDSA must establish natural ways to use distributed facilities from existing programming languages.

## 4.6. Fault-tolerance and Correctness

The demands of command and control applications have required the development of systems that are capable of continuing execution in the presence of the failure of one or more components. Distributed systems provide a natural layer at which to integrate fault-tolerant mechanisms. Furthermore, the inclusion of fault tolerance below the application layer permits applications to have some fault tolerance without any knowledge of the mechanisms or policies involved in providing this facility.

However, when we consider integrating fault tolerance mechanisms with parallelism and high latency communication, new problems arise with regard to the correctness and consistency of operations. In particular, it may be impossible to guarantee fully consistent, simultaneous views of collections of data to multiple distributed users of a database, without incurring significant performance penalties. As an example of this problem, consider the costs of doing two-phase commit transactions when the round-trip latencies are greater than the computation to be performed on the data obtained. Under these conditions, it is possible for fault tolerance mechanisms to have first-order effects on computation performance.

These problems force us to confront the following issues when addressing fault tolerance in the HPDSA environment.

*How do we provide fault tolerance in a high-latency communications environment?* Many fault tolerance schemes require the use of synchronization protocols between replicated servers. These protocols allow the servers to coordinate data updates and to provide a consistent view of objects across the entire distributed system. However, the performance of these protocols may deteriorate because of the lack of improvement in network latency compared with processor speed and network bandwidth. Therefore, our design choices for fault tolerance should take this relative increase in latency into account.

*Can we provide apparent atomicity of complex operations involving many objects?* Many applications require complex operations to appear indivisible or atomic, even in the presence of failures of any of the computers involved in the computation. For example, a money-transfer application for a bank might consider the withdrawal of money from one account and the deposit of that money to another account to be an atomic operation. If the computer maintaining the deposit account fails in the middle of the operation, there should be no withdrawal of money from the first account; a withdrawal without a corresponding deposit would violate the atomicity of the operation. While guaranteeing this atomicity of function is straightforward in applications running entirely on one processor, it is more difficult to guarantee atomicity in a distributed environment because there are more types of failures that can occur.

## 4.7. Resource Allocation

One of the purposes of a distributed system is to permit more efficient use of an organization's resources in satisfying its computational needs. For example, an organization might use idle cycles on its workstations at night to run a parallel program that optimizes the work schedule for the next day. The organization benefits directly from this application because it makes use of resources that would otherwise be wasted. However, this same strategy would be a poor solution if those workstations were to be used by a night shift of programmers. Therefore, it is desirable for organizations to have resource management facilities that allow flexibility and dynamic behavior based upon observed data. While this is not an explicit goal of the HPDSA, it is useful to consider some basic issues in resource management during its design.

*How do we optimize the use of resources for a computation?* Typically, automatic forms of resource management are done at each node by the machine's operating system. In most time-sharing and batch systems, jobs are scheduled according to priority and resource usage, and one or more operators monitor the progress of the system. In essence, the operating system is implementing a resource management policy defined by the operators.

A distributed system makes resource management more difficult for the following reasons:

- A job can be executed in many different ways (e.g., a job can be run on an expensive fast machine, an inexpensive slow machine, on several machines at once)
- A job can be optimized across more resources
- A job can be optimized according to a wide range of criteria (cost, performance, network bandwidth, total job time, etc.).

To permit effective resource management, the distributed system must supply as much information as possible regarding the needs and status of a computation while providing maximum flexibility in controlling that application.

*How do we determine the correct dimensions of the application to optimize (e.g., performance, security, correctness)?* Distributed applications can use many types of resources to accomplish their tasks. These resources include CPUs, storage devices, networks, graphics devices, sensors and specialized computational elements, such as vector and parallel processors. Each resource has a multidimensional cost associated with using it. That cost may include amortization of the purchase price of the device, maintenance cost, and consumables cost. Similarly each resource provides a multidimensional benefit to the computation, including decreased computation time, improved communication to the user, and the storage of results. Therefore, providing effective distributed resource management implies that we attempt to maximize a multidimensional benefit while minimizing a multidimensional cost. Performing this optimization requires that we understand what dimensions are most important to the users of the system, and implement resource management policies and priorities that reflect that understanding.

# 5. Architecture Description

BBN has designed a high performance distributed system architecture whose mechanisms addresses the issues noted previously. Photon's structure is designed to provide high-performance operation and data transfer while retaining an object-oriented programming model. Because Photon is object-oriented, it provides features such as abstraction, encapsulation, and well-defined client/server interfaces. At the same time, it includes abstractions that support parallelism, explicit data dependence, data streaming and redirection, and location binding to enhance performance. Finally, it provides these features in a modular and customizable way that permits implementation on a wide variety of platforms.

The following sections describe the basic architectural mechanisms of Photon. These features address the following areas:

- Layered object architecture
- Kernel-less architecture
- Multiple layers of naming
- Explicit location mechanisms
- Dynamic method binding
- Global futures
- Sequences
- Distributed shared memory
- Language veneers

Each of these features will be described in more detail in the following subsections.

## 5.1. Layered Object Architecture

The fundamental building block in Photon is the *object*. These building blocks may or may not correspond to objects in an object-oriented programming language. However, they do have two important characteristics in common with those in object-oriented languages:

- Objects are instances of *classes* that define both the data representations and operations for manipulating the data for that object.
- The implementation of objects is encapsulated and hidden from programs using those objects.

Photon extends these basic concepts to a distributed object environment. This means objects can reside on multiple machines in the environment and operations can be requested of them by any machine. An object is an entity to which program requests can be sent and from which replies may return. Objects may or may not have internal state and this state may or may not be persistent.

Our definition of objects is very general because we do not wish to limit the ways in which objects can be manipulated. A very high performance distributed system should have the flexibility to optimize how operations are performed without causing the behavior of those objects to change. For example, if the system recognizes that a program and an object it manipulates reside on the same machine, direct access to the object's data can permit object operations that have performance similar to a simple subroutine call. However, the system can only permit this type of optimization if it can still guarantee that the program performs the operation correctly and that the object's integrity remains intact. Therefore, Photon limits the amount of information that a program and an object are permitted to know about each other to that specified in a public interface description. This restriction provides maximum flexibility in the implementation of the object and permits Photon to select dynamically implementation strategies that maximize performance.

Objects are implemented in Photon by *servers*. Servers are simply programs that represent one or more objects and have a published set of operations that can be performed on those objects. Servers are the mechanisms through which objects are realized in the system; if an object has no server, no operations can be performed upon it until a server is created for it. All Photon services are provided either by servers or by code compiled or linked into the user's program. Because the built-in services provided with Photon are also provided within the object model, new services added by the user are indistinguishable from those provided by the system itself. Therefore, Photon can be considered an extensible distributed system.

Communication with objects is performed according to a protocol stack, as shown in Figure 7. Our layering model is divided into three major sections. The object layer is where most application programming is done, and includes the Application layer of the OSI protocol stack. It also includes a non-OSI layer we refer to as the Model Layer. This layer is where the concept of *proxy objects* is introduced, and permits the construction of arbitrary application interaction protocols (see section 5.10.3 for further descriptions of proxy objects). The user uses object specifications, application code, and Photon tools at this level to generate these two upper layers of our protocol stack.

The intermediate layer of our model encompasses the OSI Presentation and Session Layers. The code implementing these layers is contained in libraries which are linked with each Photon application or service program. These layers are tailored to the language and platform on which they are used.

The instance or platform layer of Photon includes the Transport, Network, Link, and Physical Layers of the OSI model. For hardware platforms with a native operating system (e.g., workstations running UNIX), the implementation of the instance layer is done in the underlying OS. This approach allows us to exploit the optimizations and tuning that manufacturers perform for native operating systems without having to reimplement it in

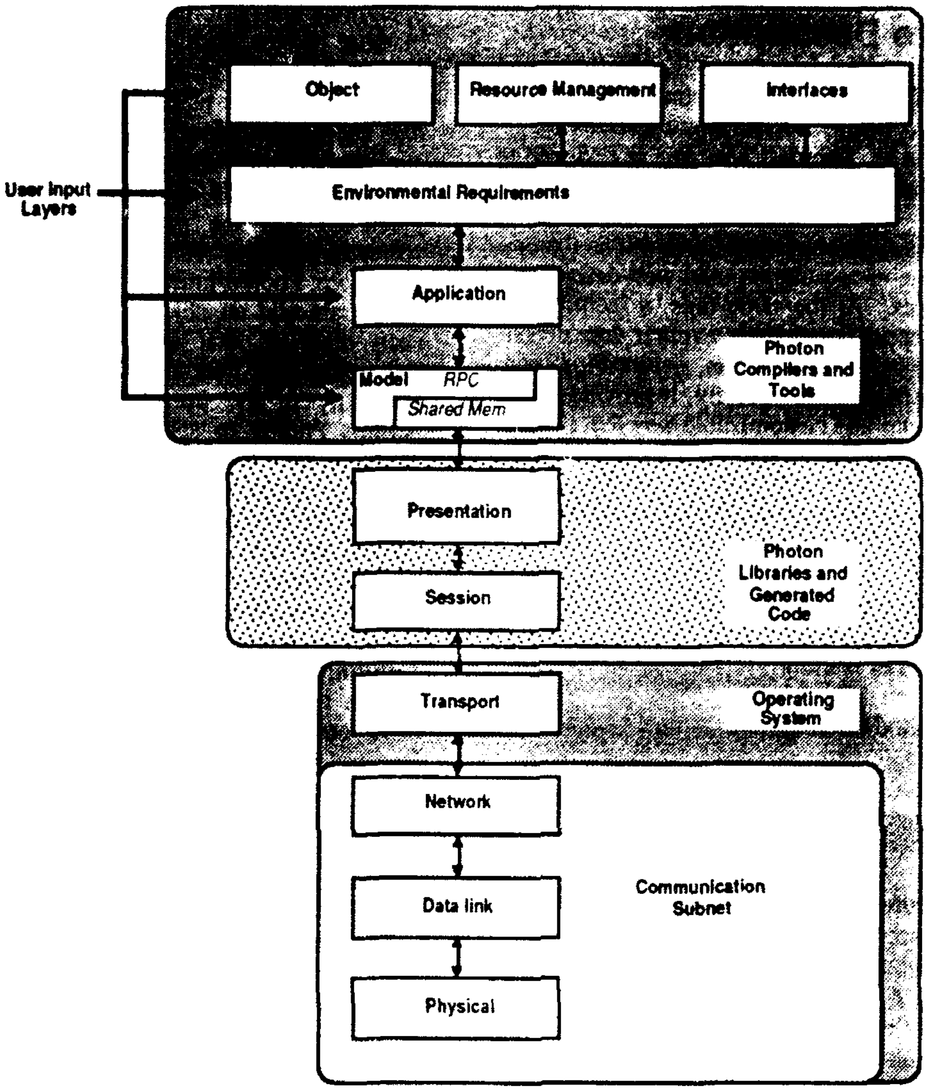


**Figure 7: Photon Protocol Stack Structure**

the distributed system. It also permits Photon to be easily ported to new architectures without the need to deal directly with the low-level hardware. However, by building Photon on top of a well-defined, simple instance layer, we still retain the ability to port the system to hardware that has either a specialized OS or no operating system.

All Photon interactions with the protocol stack are at the Session Layer (level 5) or above. Therefore, Photon can select from a variety of network protocols, local communication methods, or direct access methods in communicating with an object. By strictly obeying the layering boundary, we allow Photon the flexibility to "short-circuit" layers 1-4 when convenient. At the same time, this strict layering also permits multiple protocol stacks to be used for data transport. This ability to optimize communication strategy provides an opportunity to enhance substantially communication with objects through techniques, such as read-write shared memory.

To accommodate both object-oriented and non-object-oriented applications, we have designed language veneers that allow the programmer to control the structure of the system from the application layer. In Photon, the programmer writes language-independent interface specifications that serve as the defining boundary between client and server programs. This interface specification is then processed by Photon tools for the programming language to be used to produce language-specific interfaces for the data types and operations defined by that specification. These language-specific interfaces can then be referenced and modified by application pro-

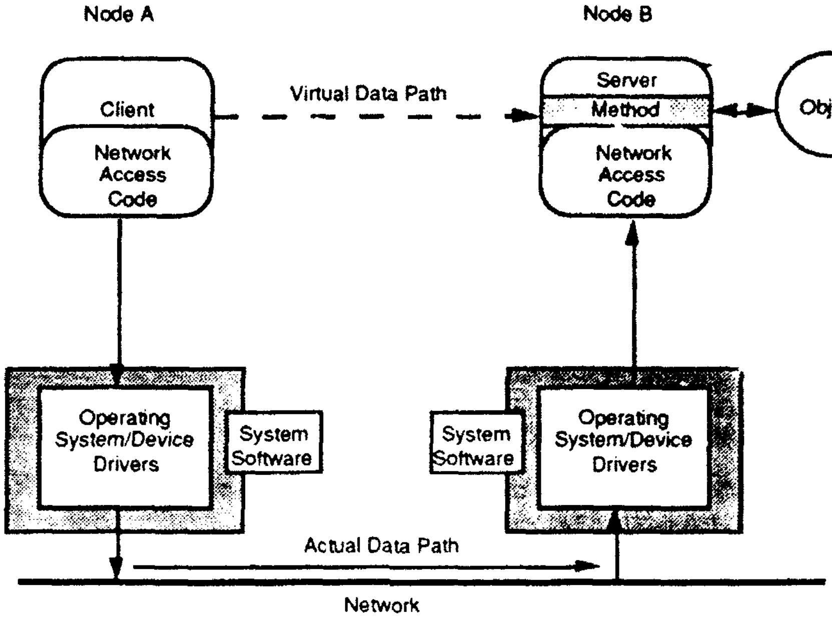


**Figure 8:** Photon **Kernel-less Implementation Model**

grammers in the supported language. This application program is then compiled and linked with the Photon libraries to produce an executable program for that platform. This program uses Photon network services to support execution.

This development paradigm is similar to that of Cronus. In essence, the Photon language tools process the interface specification to generate the application and RPC/shared memory layers of the protocol, just as Cronus genmgr translates the Cronus Object Definition Language into the target clients and servers. However, we extend this paradigm to permit the generation of arbitrary proxy objects instead of just RPC interfaces. This extension permits the inclusion of caching, replication, and communication protocols that would be excluded from a strict RPC model.

Photon also differs from Cronus with regard to how objects are related to services. While the basic addressable atom in Cronus is the object, objects are operated upon by *managers*. Managers are processes that include the methods for operating on one or more types of objects and are themselves Cronus objects. The use of managers permits the creation of very fine-grained objects without incurring the overhead of having a process created for each object. While we found this efficiency argument compelling, we felt that managers were not sufficiently abstract to be included in our model. Photon attempts to completely divorce the application from knowledge of how an object is implemented, and we believe that should include implementation details such as managers. Therefore, while Photon provides a *class* object that can be addressed for the purposes of creating new object instances, we do not include the Cronus manager abstraction. All a client program knows is how to address objects and invoke proxy objects; all implementation below that level should be invisible except for details specified in the interface definition.

## 5.2. Kernel-less Architecture

The Photon architecture provides a multiple-layer, abstract, programming model. However, this model should be implemented in a way that permits these layers to be collapsed and optimized for high performance at run time.

Our architecture assumes a kernel-less implementation which permits communication to take place directly between clients and servers without the involvement of any intermediary processes other than those inherent in the underlying operating system (see Figure 8). This structure eliminates context-switching between the Photon and application processes and substantially decreases the amount of copying of data and scheduling required for communication between client and server. It also allows the application to invoke Photon functions and routines without incurring the cost of a system call.

These performance optimizations come at the cost of increased process size. Each process in a distributed application must now include within its address space all the code necessary to do whatever form of communication it needs. We concluded that this cost was reasonable based upon the following facts:

- The application and Photon code can cooperate more closely tc optimize performance.
- Because Photor can be layered on top of an existing operating system, it can make use of virtual memory to reduce the impact of increased application size.
- The Photon code can be implemented as shared libraries, further reducing requirements on main memory
- Running the Photon code in the user's process and at the user's privilege level preserves the integrity of the underlying operating system's security mechanisms.
- All aspects of Photon operation are accessible and visible during application debugging and testing.

While kernel-less implementation is not strictly required by the Photon architecture, the performance, debugging, and security advantages recommend its use.

## 5.3. Multiple Layers of Naming

Photon includes multiple levels of naming. These include symbolic names, location-independent or logical names, and location-dependent or physical names (see Figure 9).

Symbolic names provide methods of addressing services that can be understood easily by people. Symbolic names are usually alphanumeric strings such as `MandelbrotComputeServer`. Symbolic names are converted into other types of names by a *name service*. The name service is implemented as a collection of servers that can be accessed by any Photon client. The function of these servers is very simple: given a symbolic name, they return a location-independent name.

Location-independent or *Logical Object IDs (LOIDs)* are used to identify a service without being specific regarding its location. LOIDs are fixed-length structures and are not necessarily easy to interpret other than by machine. However, they do provide a unique identification for a service throughout the Photon system and this identification can be either stored or used to access servers through the *locate* operation. Note that LOIDs can be resolved to more than one object or server. Protocol stacks supporting multicast addressing may wish to associate multicast addresses with LOIDs.

Location-dependent names or *Physical Object IDs (POIDs)* are names that specify the physical location of an object in Photon with respect to the underlying communications network. Use of POIDs allows the system to associate an object with a communications link or connection and thereby optimize the communication over that link. These names are bound to the underlying communications protocols the application wishes to use.

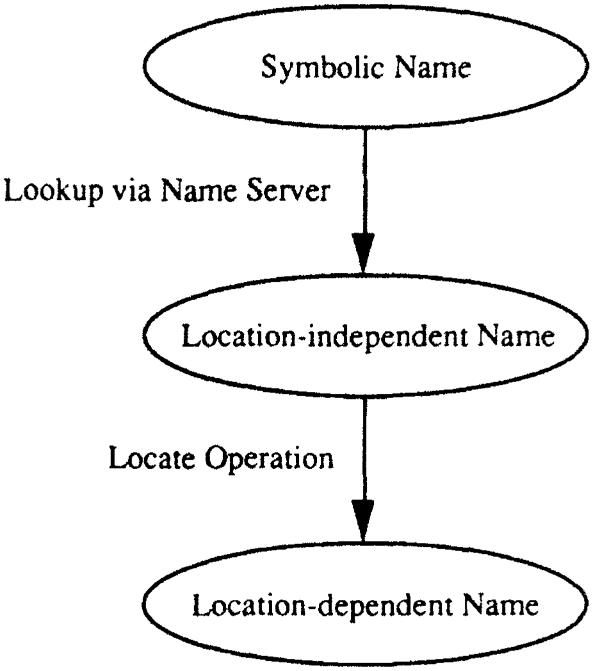


**Figure 9: Types of Photon Naming**

## 5.4. Explicit Location Mechanisms

Photon requires that a process know the location-dependent name of an object before it can perform operations upon it. The location-dependent name is discovered by doing a *locate* operation on a location-independent name. This means that even when an object may be replicated on many nodes, the system will select only one of these objects at the time a client does a locate operation. The client will continue to use that instance of the object until there is a reason not to (such as an exception).

The use of an explicit location operation rather than a transparent, dynamic rebinding at a lower level (e.g., Cronus's broadcast locate) has the following advantages:

- The locate operation establishes a binding between the client and the server. By establishing a binding, we provide the system with a way to reference this particular communication path and a reference point for exception handling.
- The locate operation allows the client to establish a location-dependent communications link to the server. This permits optimization of the communications path for that object.
- The communications needs of the application, such as latency or bandwidth constraints, can be negotiated with the servers and network layer at the time of the locate operation. Should the underlying network require reservations or the setup of a virtual

| | | Reduces |
|---|---|---|
| Data Driven | Pipelining | Total Execution Time |
| | Data Flow | Unnecessary Blocking |
| Demand Driven | Data Flow | Excess Computation |
| | Shared Memory | Excess Data Transfer |

**Figure 10: Interprocess Communication Mechanisms and Their Effects**

circuit, this can also be done at this time.

- If a virtual circuit is used, authentication of both the client and the server can take place once at the time of the locate rather than on every operation on the object.
- The application has an opportunity to modify its behavior based upon the location and communication abilities of the object found.

The largest disadvantage of explicit location is in the area of fault-tolerance. Because we have explicitly bound the client to the server, rebinding after the failure of the server is not transparent; when a failure occurs, a new binding must be negotiated by the client. While an unsophisticated application may wish to be shielded from this information, a sophisticated one might wish to take action based upon the failure. For example, an application sending data to a replicated database might wish to do more batching of its data when sending to a copy of the database over a high-latency link.

In general, explicit location will not be seen by the programmer at the object level; Photon proxy objects will often be capable of performing this binding transparently to the client programmer. However the lower levels of the system will use explicit location to acquire location-dependent names, and therefore the advantages of the technique will be available to the sophisticated application programmer and object designer.

## 5.5. Parametrized Communication

The relationship of Photon communication mechanisms to concurrent distributed programming paradigms, traditional object oriented systems and physically shared memory and distributed computing environments is illustrated in Figure 10.

One of the goals of the HPDSA is to promote concurrency while supporting data flow, demand-driven, pipelined, and shared memory programming models. Data driven models strive to reduce execution time by initiating concurrency in response to the availability of results and reducing idle time spent on synchronization. In these approaches, execution is started in response to either regularly arriving results (pipelining), or asynchronously arriving results (data flow). The data source and data processing functions may execute concurrently with each other and with the communication process, thereby improving parallelism. Demand-driven approaches strive to reduce the amount of resource required to perform a distributed computation. In demand-driven data flow, inputs are computed and supplied only when a processing function requests them. In shared memory, computations execute concurrently and values are transmitted as required.

In Photon, this range of paradigms is supported by a uniform set of communication abstractions and an object oriented, functional style of programming. Shared memory objects allow data values to be published to other processes (see section 5.9). Updates to these values may be data- or demand-driven, depending on how the methods for the object are implemented. However, the application program making calls on the shared object need not be modified. The method implementations can also be written to adapt as delay characteristics change. Futures and sequences allow one client process to combine servers to perform a computation, with one producing data and the other processing the values as they are generated. They allow

| Application Demand | Communication Approach |
|---|---|
| Unreliable, on demand requests | Datagrams |
| Reliable, sequential transmission | Connections, atomic actions |
| Guaranteed transmission rate | Reservation protocols |
| Sharing policies | Allocation and fair queuing |

**Figure 11: Communications Approaches for Various Application Requirements**

values to be communicated directly between the servers, without incurring additional communication or process scheduling delays that occur with current RPC when values are passed through the client. Language veneers and proxy objects extend the current RPC notions of automatic RPC stub generation to allow programmers to export code and data along with caching strategy and other algorithms to the client.

There are several types of communication requirements that might be specified by an application as shown in Figure 11. The Photon architecture offers two ways in which the choice of communication attributes can be made. In the simpler approach, attributes can be applied to individual requests. For example, a program can mark a particular future, sequence or memory object as unreliable or reliable, or associated with a particular communication bandwidth for data streams or regular updates.

Photon servers may also offer different physical object identifiers (POIDs) for a given object, each with a different set of service properties. For example, one POID might be used to support unreliable datagram requests and another for establishing reliable connections. Some connection POIDs might also support bandwidth reservation, which would be marked on the property list.

The association of property lists for OIDs is a promising framework for handling resource management. For example, we might allocate a particular bandwidth for communication with a class of objects. Property lists would indicate when this was possible, and special POIDs would be used to reference these objects via the reserved communication class methods.

## 5.6. Dynamic Method Binding

Photon's explicit object location and kernel-less architecture permit the use of a technique called *dynamic method binding*. Dynamic method binding is a mechanism through which the server can instruct the client about what routines should be used for performing operations on an object it represents. This mechanism is in keeping with our policy of the client specifying what needs to happen and the server specifying how it is to happen. This technique can be used to optimize the communications path between the client and the server without requiring direct involvement by the programmer of the client.

Dynamic method binding is invoked as part of the location of an object. When a client is bound to a server by means of its physical name, the server engages in a negotiation with the Photon support code in the client to identify the best communication paths between the two processes. The server then instructs the client about which compiled-in Photon routines to use to achieve this communication. The client code stores that information and uses it to select the optimal code for invoking the object methods.

This technique was introduced in Photon to permit the same program to be run in both distributed and parallel environments without sacrificing performance in either. Figure 14a shows the desired program structure for an application running in a distributed environment. In this case, a client and a server are on the same local area network; therefore, the client uses local area network communications routines to invoke server methods that operate on object X. However, when this same application is run on a shared-memory parallel processor, the program structure shown in Figure 14b is more appropriate. Because both the client and server are executing on the same computer (albeit on different processors), they can use shared memory for their communications path. This eliminates the operating system from the communications path, thereby significantly improving the communications performance.

Ideally, one would implement dynamic method binding by having the client and server dynamically link their communication routines at the time the object is located. While this is an elegant and general implementation, the facilities to implement true dynamic linking are not present in many existing compilers, linkers, and operating systems. However, true dynamic linking is not absolutely necessary to provide dynamic methods. If we assume that an application is linked with libraries for all appropriate communications methods for a given platform, then dynamic method binding only requires the se-

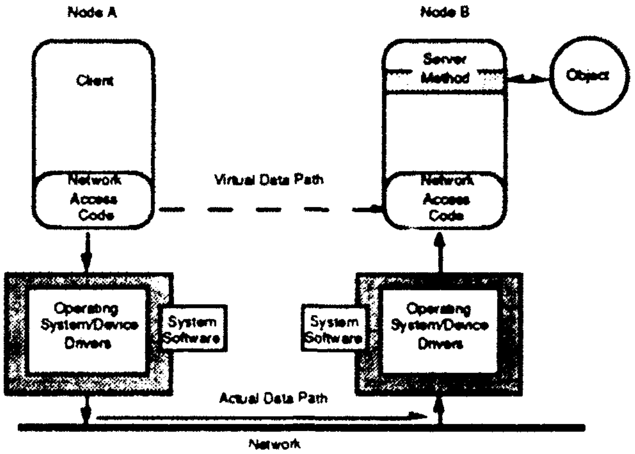


Figure 14a: Communications Routines for Network Communications

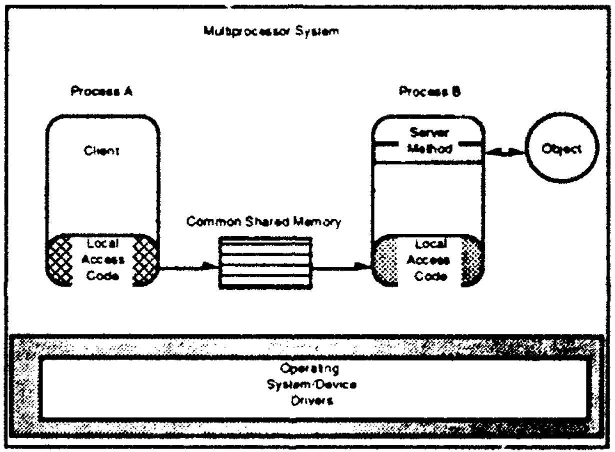


Figure 14b: Communications Schema for Interprocessor Communications

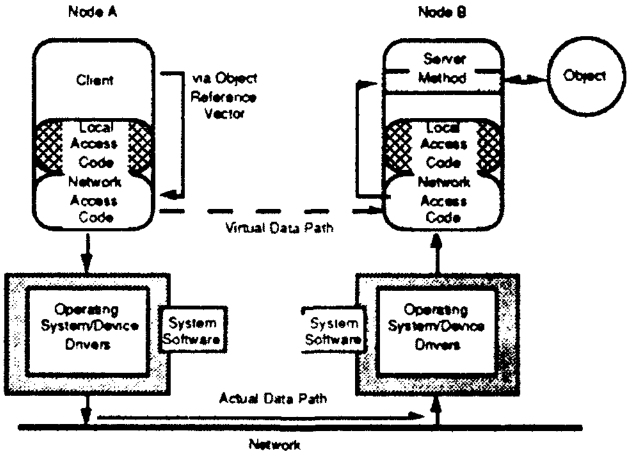


Figure 14c: A Simple Implementation of Dynamic Method Binding

lection of which library to use at run time (see Figure 14c). This selection can easily be done by means of a modifiable dispatch table and requires very little assistance from the compiler, linker, and loader. While the application will be larger as a result of including all the communications libraries for a given machine, the use of virtual memory minimizes this impact on main memory. Those libraries that are never referenced will never be paged into main memory, and therefore the working set of the actual running application will not be significantly different from that of an application that was bound to only one set of libraries at link time.

## 5.7. Global Futures

Many techniques exist for allowing a single process to start and control several concurrently executing activities. These include multitasking, futures, and streams. Multitasking approaches include the concept of threads implemented by Mach [Rashid Mach]. In multitasking systems, each thread may be blocked waiting for a different remote request. When the request is finally satisfied, thread execution continues, guided by normal scheduling policy. *Futures* include the approach taken by Cronus [Walker Cronus futures]. Rather than blocking RPC requests until the value is returned, the request returns a *future* which can be claimed when the value is needed. A thread may have several pending futures, and may invoke requests that create more. Futures may be combined into a *future set* which can then be used to block the thread until one of the set's values has been received. *Streams*, such as those implemented in Mercury [Liskov Mercury], allow a client to submit a sequence of requests and then claim the replies in the same order as the requests were given. This allows client and server to execute in parallel. This range of approaches is generally adequate for distributed processing driven by point-to-point requests between a single client and many servers.

The *global future* is an addition to the basic RPC/call stream model. In the global future model, a caller may issue remote procedure calls to various servers and receive the results of these calls at a later time. The asynchrony inherent in the future mechanism allows the client to issue many calls before receiving the results of the first; this allows computation in the server to proceed in parallel with computation in the client since the client does not have to wait for results from one call before issuing a second call.

Global futures extend this capability by allowing the client to issue calls to several servers, with output from a call executed by one server being transmitted directly to a second server. This forwarding of data allows computation in the server to proceed without the client being a bottleneck between them. The computation in the servers may also proceed in parallel.

A future is a typed value. It has a globally unique ID. When the client directs the server to perform a computation, it issues an RPC, for example:

```
ObjID.operation(input_param_list,
                output_param_list)
```

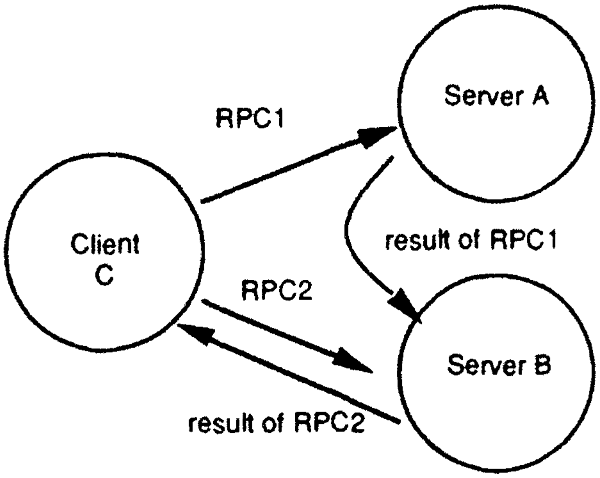
In a remote procedure call without global futures, only the values of the input parameters would be communicated to the server, and later the values of the output parameters would be sent back to the client by the server. Only then could the client supply the value received as an input parameter to another RPC call to a different server.

By using global futures, the client can cause a result to be forwarded directly from one server to another without its passing through the client (see Figure 12). The client does this by specifying the ID of a global future in place of one or more of the input or output parameters. When a future is supplied instead of an output parameter, it will usually specify a distribution list. The distribution list directs the server to send a copy of the output value to each of the destination hosts in the list as soon as the value is computed. When an input parameter is specified as a global future, the server waits for the value to arrive, binds the value received to the parameter occupied by the future, and then proceeds with the operation requested. Since future IDs are allocated by the client without consulting the server, a client can set up a flow of values from one server to another with no waiting for intermediate results. As a result, the joint computation between the servers will complete as soon as possible.

### 5.7.1. The Futures Port

We have defined an abstraction in Photon called a *port* that is a unique identifier for the source or destination of a future value. This is unlike the TCP protocol, where a port is a host-id, port-number pair, and a connection is uniquely identified by two host-id port-id pairs. In our architecture, we have arranged that just the port identifier is sufficient to identify the source or destination of a future value. While the connection between the server and the client is a simple two-way pipe with data flowing between two ends, the futures port may communicate with multiple peers. Thus, given a client and a number of servers, there the possibility of connections are possible between each pair of servers, as well as with the client and each server. Whether the actual operation of these connections is datagram-like or connection-like is a function of each underlying protocol.

Layered above whatever protocol is used for transport is the Photon inter-object protocol. Protocol data units in this protocol are called *directives*. When a Photon peer wishes to communicate with another peer, it sends one ore more directives to that peer. An actual connection will be established as necessary, using one of the available transport protocols, or datagrams will be used in one of the available protocol stacks. Directives



Client wishes to execute:

```
RPC1(A, &Result1, Noinput);
RPC2(B, &Result2, Result1);
```

Client creates a future for A
with a destination of B

```
f1 = future(A, B)
RPC1(A, f1, Noinput);
RPC2(B, &Result2, f1);
```

Resultant data is transferred
from A to B without C's
involvement

**Figure 12: Direct Forwarding of Global Futures**

are used to invoke methods, provide the value of a *future, delete futures, and perform a number of other* functions.

### 5.7.2. Garbage Collection of Global Futures

A global future must be held by the producer of a result until it is no longer needed. We use a system of distributed reference counts to accomplish this. (Note: the server which produces a future will sometimes be called the producer, and the various servers to which the futures are directed called consumers.) When a future is produced, it is held in a buffer with its reference count. This reference count is normally set to the number of destinations to which the future is to be sent, i.e. the length of the distribution list. A copy of the future is sent to each consumer, and as each consumer acknowledges receipt of the future, the reference count is decremented. When the reference count reaches zero, the producer may delete its copy of the future. This type of reference count is called a source reference count.

There is also a destination reference count. When the future is produced and forwarded to its consumer, a destination reference count is included in the message. When the consumer receives the future, the consumers local reference count is set from this value. Each subsequent use of this future as an input value to an RPC invocation will decrement this reference count, and when the reference count reaches zero, the copy may be deleted in the consumer's memory. The destination reference count is supplied by the client as part of the distribution list. Destination reference counts are most commonly set to one.

Reference counting is suitable for a large class of distributed applications, but not for all applications. If a consumer needs to use a particular global future an undetermined number of times, its destination reference count can not be set ahead of time in the client's RPC invocation. Distribution lists also have a similar problem—the set of recipients of a future may not be known at the time of the invocation of the RPC. We handle these cases with a slightly different mechanism.

The destination reference count of an item may be set to -1. *This indicates that the result should be held for* an indefinite number of claims. In this case, the client is responsible for knowing when the destination's copy of the global future is no longer needed. When this condition is true, it sends a `delete-future` directive to the consumer that is holding the copy of the future that it no longer needs or to each consumer which has such a copy. The `delete-future` directive causes the copy of the future to be deleted regardless of its reference count.

The source reference count may similarly be set to -1 to cause the copy of the future to be retained by the producer indefinitely. Once again, the client assumes responsibility for the deletion of the future when it is no longer needed. This is reasonable, since only the client may cause the future to be sent to consumers—i.e. the client knows when it will no longer issue any RPCs which will require the future as input. There are several reasons a client might direct a server to forward a value well after it has been produced, even though in this case the client could send this value instead of the future. This could happen if:

1) the client does not want to store the value,
2) it may be sending directives to the two streams well ahead of execution time, or
3) one or more such futures may be so large that it would consume considerable bandwidth to forward them.

### 5.7.3. Keeping Track of Live Global Futures

Although the client's application program could be made completely responsible for producing the `delete-future` directives, the Photon library automates most of the hard work. This is done by keeping a record of the set of source and destination futures with their reference counts in the clients memory and by shadowing those records in the server's memories. This mechanism works as follows.

When a future is allocated, a record is made by the library of the existence of this future in the *future list*. This entry holds two reference counts: a copy of the remote reference count and a local reference count set initially to 1. The entries in this list are referred to as *future list entries*. For simplicity, first consider an entry for a future whose remote reference count has been set to -1, indicating it has been created for indefinite use. The first time the future is used as an input parameter to a particular consumer, a `transmit-future` directive is sent to the producer, and the future list entry's reference count is incremented by 1. An entry in a *copy list* is made to record that a copy of this future will appear at the destination's memory, and an entry in a *RPC list* is also made to record the RPC that will consume the future. As RPC completions flow back to the server, they are matched up with the appropriate RPC list entry and will then cause the one or more copy list entries to be found and have their reference counts decremented.

When the client knows that it will not issue any more RPC invocations with a particular future ID as input, it executes a `release-future` call. This call decrements the local reference count of the future and copy list entries, but does not necessarily delete the future list entry because the reference count may not have reached zero yet. When the last RPC completion referencing the entries also comes back, `delete-future` directives can be sent to the various holders of copies of the future.

### 5.7.4. Error Handling

We must consider how to handle garbage collection of futures in the face of errors. For example, suppose a stream of RPCs is being directed to a server B, which is supposed to consume futures being produced by server A. After a while, an RPC to server B fails, and some RPC's which were in the pipe after the error are not executed. As long as our client's connection to server B's port remains open, recovery is possible.

Assume now that the client hears of the RPC failure, and knows that RPCs that were in the pipe after the failure will not be executed. For items with indefinite reference counts, the reference count structure maintained by the library in the client's memory allows Photon to go through and decrement the future list reference counts as if the skipped RPC's had completed. Thus, the application can proceed, possibly issuing other directives or RPCs to server B, knowing that extra copies of futures will not build up in server B's memory.

To handle errors involving items with finite reference counts, the library must also maintain a similar reference count structure. The difference is that a `delete-future` directive does not need to be sent to the producer when its remote reference count goes to zero -- instead the future list and copy list entries can be simply removed. When an RPC fails, this structure is sufficient to enable the library to know what futures will never be consumed and issue `delete-future` (or `decrement-reference-count`) directives to see that they are deleted. When a server fails completely (indicated by sending a RESET), resources in other servers that are tied up can be cleaned up by the client. A server that receives a RESET from its client can clean up all resources allocated by that client. A server that receives a RESET from one of its peers can decrement the reference count of futures waiting to be acknowledged by that peer.

### 5.7.5. Race Conditions.

We consider three kind of race conditions.

*Race 1: A future arrives for a port which does not exist yet.*

This condition is prevented by the rule that a client may not construct a distribution list which contains a destination to which it does not already have a connection.

*Race 2: A future is needed from a port which has closed.*

The various reference counting mechanisms in the client and in the servers will assure that this condition does not occur. When a client closes down a connection, the close is placed in the pipe after the last RPC to the corresponding server. The connection is not actually closed until all RPCs and directives to that connection have been executed. So if a port is closed, it was done at the behest of the client. In this case, a RESET will be returned, and the source can treat this as an acknowledgement.

*Race 3: A consumer tries to execute an RPC for which the future has not been produced yet.*

In this case the server waits patiently for the future to arrive. It could poll for the value, but if the client application is working properly, the value will be forwarded as soon as possible anyway, so there is no reason to poll. The consumer can poll with a hint if it has never before heard from the producer before in case the producer has backed off retransmission to the minimum rate.

### 5.7.6. Handling Communication Failures

We require our connections, or association, between client and server to be maintained indefinitely. By this we mean that connection halves do not go away just because communication has apparently halted. Instead, retransmission is backed off to a slow rate, but state is retained. A connection is considered broken only when a RESET is received from the other end indicating that the connection at the other end really does not exist.

### 5.7.7. Forwarding a Global Future to a Second Level Consumer.

Suppose a piece of server code (in server B, say) which was supposed to be a consumer of a future wanted to treat a future like a data item, and pass it on to another server D. The client would not be able to reference count this future properly. One solution is for the server B to form its own future, which would forward the data value to D as soon as it arrived at B. A second solution is for B to send a `transmit-future` command to the source of the future. It can do this provided it has not acknowledged receipt of the future; once it acknowledges the future, A may no longer have a copy. Server B would also have to delay reporting completion of the RPC in question until it knew of the completion of the RPC on D which was going to consume the future. If B has received the future, it can forward the value instead. The disadvantage of this latter method is that the client no longer knows exactly what is going on. Instead, the forwarder, B, must take on responsibility for cleaning up any downstream communication errors.

## 5.8. Sequences

In Photon, the user sees no explicit concept of a network connection. Instead a client program can invoke operations specifying that the output of one operation is to become the input to another. Inputs and outputs from operations are typed, and Photon's type system includes features that effectively provide a connection as a data type.

Photon's type system allows the same set of canonical types to be defined as Cronus, with two extensions. The two extensions are the *tagged union* and the *sequence constructor*. These two extensions work together to handle the case of a large flow of data between twc processes or threads executing in parallel.

A *tagged union* is a structure that contains a tag and a union, with the values of the tag specifying which element of the union is present. This is effectively the same concept as the record variant in Pascal. An application program is required to access the element of the union specified by the tag—otherwise its behavior is undefined. Therefore, the tag acts as a discriminator among the various member elements of the union.

To illustrate, the proposed Photon Object Description Language (ODL) syntax for a tagged union is:

```
cantype C_rec: union,
        tag: integer a,
        choices are
            1: integer i,
            2: B_type b,
            3: C_type c
    end;
```

The case labels (1, 2, and 3 in the above) must be values of the same type as the tag. Note also that the tag is a labeled element of the union which can be referenced by its identifier.

The *sequence* mechanism is used to transmit a pipeline (or queue) of values between two processes or threads in Photon. A sequence of any data type can be defined and used, like a future, as an argument to a method. Sequence types may be used as arguments to methods but are not first class data types because they may not be used as fields of other data types. The value of a `sequence of T` is an ordered set of values of type `T`. The sequence is constructed by `put` operations, and accessed by `get` and `next` operations. The `get` operation returns the current element. The `next` operation advances the sequence past the current element so that the `get` operation will then return the next element in the sequence. The sequence may also be terminated, at which point the `get` operation will indicate that all the elements have been read.

## 5.9. Distributed Shared Memory

Photon borrows and extends the distributed memory model of Munin [Bennett Munin]. In this model, locations in a global memory space, called *memory objects*, are globally addressable, variable size segments that have a *memory type*. The type of a memory object corresponds to the caching strategy used for it—including the algorithm that controls distribution and invalidation of updates. The caching strategies vary based on how consistently and reliably memory is maintained, and how classes of reference are matched to the characteristics of multiprocessor, local, and wide area network access. Photon memory objects are effectively light weight objects that may be used to accomplish shared-memory-style distributed programming below the level of RPC and may serve as the underpinning for the other Photon objects.

Formally, a Photon memory object is a tuple of the following form:

```
<memory-ID, memory-type, memory-type-info,
 memory-object-state, representation-type,
 representation-value>
```

and a memory-ID is a tuple of the following form:

```
    <producing-host, controlling-host,
     unique-number, sequence-number>
```

In some cases, depending on the memory type, producing-host and controlling-host may be the same, or only one of the two may be used.

A memory object carries a representation value which may be written, read, mapped, and modified but which does not in any way interact with its memory type. It also carries some state associated with its type (e.g., locks, "dirty" flags, etc), and may carry other descriptive information specifically related to its memory type. A Photon memory object carries a representation type that is distinct from its memory type. The representation type defines how the contents of the representation value are mapped into the data representations of various machine architectures. This roughly corresponds to the data structuring capabilities of the Presentation Layer in ISO/OSI. In adding the representation type to the memory type, Photon goes beyond the Munin memory model. In the Photon ODL one may define types and metatypes. Types are Presentation Layer types described above. A metatype corresponds to a representation type paired with a memory type.

A memory object is one kind of Photon object. However, a memory object may or may not be made available for general access. A memory object may be used to hold the representation of another Photon object (which is not a memory object), and in this case, only the higher level Photon object would have direct access to the memory object which holds its representation. A Photon object may also use other means to hold its representation if the designer so wishes. An example would be a Photon file object, which would store its contents in disk storage rather than main memory.

### 5.9.1. Addressing of Photon memory objects.

A Photon memory object is addressed by its memory ID. A memory ID maps directly to a Photon OID, so that the set of Photon memory IDs is a subset of the set of Photon OIDS. A program or executing an operation upon an object may, theoretically, map a Photon memory object into or out of its virtual address space. In practice, the ability to map Photon memory objects may be unavailable or limited due to nature of the local operating system's virtual memory system. However, a non-mapping style of interface to the interface using get and put operations is always available and can always be used to invoke any capability of the Photon global memory system.

A Photon memory object must be created before it is used. This is done by specifying a piece of the memory of an active process containing a local representation of the memory object's initial contents. When this operation is performed, the caller must supply a pointer in its virtual address space to the copy of the object, an indication of the Photon memory type, an indication of the representation type, and possibly other parameters. Under the mapping style of interface, this memory area would become the mapped image of the actual Photon memory object. Under the get-put style of memory object interface, the area would only supply the initial contents of the memory object —i.e. creating a memory object is like creating it with undefined contents and immediately thereafter issuing a put call to set its contents.

### 5.9.2. Read-Write Semantics.

A Photon memory object may be mapped into the virtual memory space of a process, provided it knows the memory ID of the object. Once mapped, the value of the object may be read at the mapped range of locations. The process may or may not modify the object (write the locations), and modifications may or may not be seen by other processes mapping the object, depending on the memory object type and type of mapping operation performed.

### 5.9.3. Memory types and Caching

Copies of a Photon memory object may be located in more than one Photon host machine. The copies are considered to be in the local cache of the Photon memory system. This would occur if processes in different machines mapped the item simultaneously. When this occurs, the processes must obey some discipline in referencing and updating the object, just as would be the case if several processes mapped a piece of shared memory in a multiprocessor. A typical discipline is that only one process will modify the data at a time, and the processes mediate the right to modify the shared area by acquiring a lock. Another common situation is that one process always writes the data, and all others only read it. The various Munin memory types were chosen to match the actual patterns of access exhibited by shared memory programs. Each Munin memory type obeys a different cache update strategy suitable for its usage pattern. Photon includes Munin's set of cache algorithms, and extends it by making it an extendible set—in particular, various data replication strategies, for example, version voting in Cronus, are modeled as cache update strategies.

### 5.9.4. Synchronization

Synchronization between copies of a memory object is a function of the memory type. The producer-consumer memory type knows that a value is only written by the creator and then subsequently read by various consumers. Thus no special effort is required to keep copies identical—i.e., synchronized. However, since computations will proceed faster if values are forwarded in advance of their being needed, and copies need to be deleted when they have been consumed, the producer-consumer type uses a system of distribution lists and reference counts to implement a policy that assures that values rendezvous with consumers and are deleted as soon as possible. This rendezvous process, of course, constitutes a form of synchronization between processes. At the RPC level, rendezvous with a producer-consumer value in the memory system is called *claiming a future.*

Other memory types use different algorithms for synchronization of values in different caches. For those memory types modeled on cache coherency algorithms, there is nothing new here. Replication memory types use their distinctive algorithms to effect reliable replication of memory objects.

Photon will initially use the version voting method of Cronus as its reliable memory method. Other types will be able to be implemented in the Photon library, and users will be able to extend the set of memory types by defining memory type servers.

### 5.9.5. Locking

In Munin, a special class of memory objects is the lock class. Whereas all other types of memory objects are accessed with `map/unmap` or `get/put`, locks are accessed with `lock` and `unlock` primitives. This is because the algorithms for obtaining and releasing locks are similar to those for obtaining, updating and invalidating copies of memory objects. Secondly, given sufficient hints, the memory system can optimize the propagation of updates by noticing where locks are held and when they are released. For example, if the Photon memory subsystem knows that a set of memory objects are locked by a particular lock, then it can defer broadcasting updates from system holding the lock while the objects are updated, instead of sending updates when the lock is released. For simpler cases, the memory object will be able to serve as its own lock. Thus, a set of simple locking operations will be defined as memory access methods that apply to any memory type—unless the memory type specifically overrides the method, making it invalid.

### 5.9.6. Representation type

A Photon memory object carries both a memory type and a data or representation type. We have described the use made of the memory type above. The representation type makes it possible for a memory object to be shared between systems with differing machine architectures—i.e. different word sizes, different floating point formats, etc. When a copy of a location is sent to a peer memory system in an update, if the peer is not of the same architecture, the contents of the object is converted to a machine independent representation which is placed in the message sent to the peer. The receiving peer converts the machine independent representation to initialize or update the contents of its copy of the memory object. When practical, Photon peers may negotiate away the use of the machine independent representation. This permits the substitution of native representations if the peers are on machines of compatible architecture, or inexpensive representations that both machines are capable of producing and parsing, if such exist. Photon makes no use of the local representation of the memory object, but makes it available to the application level.

### 5.9.7. Access Control

Some sort of access control clearly belongs in the memory level of the Photon design. We propose no new concepts in the access control area. If access control is incorporated at the memory level, it would be an independent property of a memory object—"orthogonal" to memory type and representation type. Access control could also be incorporated into the object level—i.e. in heavyweight objects, by inheritance from one of a number of defined objects which in turn inherit from a general access control object base class. At present this is an open issue in the Photon design. The simplest approach would to adopt the access control scheme from Cronus and insert it into the memory level of the architecture. This corresponds to the latter scheme mentioned above.

## 5.10. Language Veneers

Photon is designed to support languages such as C, C++, and Ada directly with libraries of support routines, definition files, macro, class, or generic definitions, preprocessors if appropriate, and tools to generate libraries, header files, and other code for Photon classes defined in the Photon Object Definition Language. Photon may support other languages. Access to Photon facilities can be provided to any language, although there may be practical reasons for not doing so.

In this section, we describe our design for Photon's support for the C++ language. We emphasize C++, because it is well matched to the object oriented concepts of Photon, and it allows us to explore distributed programming in a more modern context than the C language. Because C++ provides sufficient definition facilities, macros and specialized preprocessing are not necessary. In effect (at least for our purposes), C++ is an extensible language. Photon objects are mapped into C++ objects that are either actual implementations of the objects or proxy objects that send messages to the Photon class or object server requesting that the corresponding method be performed on the actual object.

Photon's language support for the C language generally follows the model used by Cronus. Structures are defined to represent objects, and routines are defined to operate on those structures, but the C language is is not extended.

### 5.10.1. Photon Object Definition Language

Photon provides the Photon Object Definition Language (ODL) which may be used to define classes. Classes are implemented in programs which act servers for objects of the class, and are accessed by programs, known as clients. A program may be both a client and a server, even for the same type of object.

The Photon Object Description Language is an extension of the Cronus object description language. It is deliberately unlike C, PL/1, ADA, etc. in an effort to be language neutral -- that is, equally appealing to users of various languages. ODL defines *cantypes* which are units of structured data that may be accessed or passed through Photon. It also defines classes, and for each class, the methods and parameters that may be applied to a member of the class. Any parameter to a method may be a cantype, and the state of an object may be represented by a cantype. If a cantype T is defined, Photon automatically defines Future<T> and Sequence<T> -- these are sometimes called metatypes.

The designer of a Photon application designs one more classes by writing class descriptions in the Photon Object Description Language. A file containing this description is processed by the Photon Description Language processor. It may be processed only for syntax checking, or to create and/or register a class descriptor (assuming it finds no errors). A class descriptor is itself a Photon object. Once a class descriptor exists, the processor may also be used to produce language specific description files for inclusion in programs which reference objects of the class described. These files contain declarations of routines, structures, classes, templates, and possibly macros, depending on the target language and on the class itself. At the present time, the Photon Object Description processor is the only language preprocessing we expect to use in Photon.

### 5.10.2. Mapping ODL into Objects in the Target Language.

The class description is mapped into declarations in the target language. An object has a state, which is held in a cantype declared in the representation clause of the description. The cantype is a structure in an abstract data space, but it can be translated directly into a data structure in each supported target language. In Photon, this translation can happen both dynamically and statically. The static translation occurs when the language-specific description file is produced from the object descriptor. Effectively, the Photon object description is translated into a description of the object in the target language. The translation would occur dynamically when the representation was passed between two servers for the same class which were written in different languages—or between two client programs which operate directly on the objects.

Photon uses an extension of the Cronus abstract data space. Cronus and Photon use their abstract data spaces in the representation clause of the ODL, and in defining the type of parameters to methods. One extension to the abstract data space is the tagged union. Other basic types and structuring concepts are identical. In C and C++, a tagged union is mapped into a structure containing a tag and a union. The tag is mapped according to its type, and the elements of the union are each mapped separate according to their individual types.

Metatypes are also mapped. In C++, the metatypes can be mapped directly, using templates. The metatype sequence is mapped by the Sequence<class T> template, and the metatype future is mapped by the Future<class T> template. Either of these templates may be used for any type T whether or not T is defined within ODL or simply defined in the local program, provided a program is operating within the Photon environment. Such a metatype could only be used inside a single program. However, distributed programming through Photon requires that the base type of the metatype be a cantype declared in ODL.

In our work on Photon, we have prototyped templates by using macros since the version of the C++ compiler we are using does not yet implement templates. However, templates are standardized in the C++ language and due to appear soon in most C++ implementations. The latest version of *cfront*, a C++ to C translator from AT&T, includes the template feature. The macro approach suffers from some clumsiness and is difficult to debug.

### 5.10.3. Proxy Objects

One of the key elements in our object-oriented language veneer is that of *proxy objects*. Proxy objects are class definitions that represent remote objects within a client program. These class definitions are created by the class designer to work cooperatively with servers of that object class. Proxy objects permit the class designer to implement arbitrary caching and consistency policies between clients and servers. Figure 13 shows a schematic representation of how proxy objects can be used for communication between client and a server.

Remote procedure calls are simply a special case of proxy objects. RPC stubs generated from an ODL are

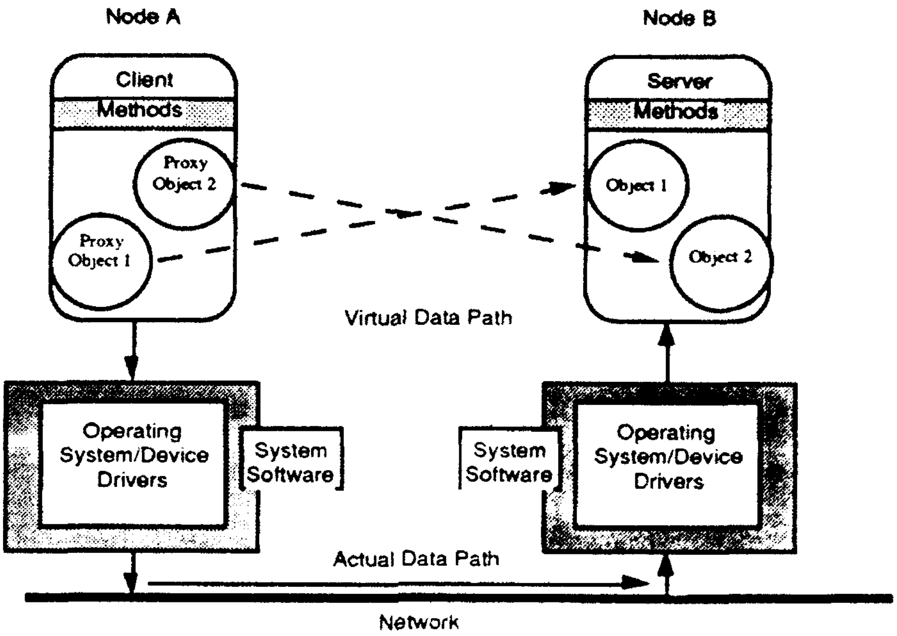


**Figure 13: A Schematic Representation of Proxy Objects**

merely proxy objects that send a message to the server for each operation requested upon an object. However, it is not difficult to visualize object definitions where these trivial proxy objects cause unneeded communication. For example, imagine that there exists a class `ball` that has operations `define_color` and `read_color`. A client might contain code such as the following:

```
color c;
ball b;

b.define_color(red);
...
c = b.read_color();
```

Using an RPC proxy object, these two operations would require two separate messages to be sent to the `ball` server, and two replies to be received. However, a more intelligent proxy object might simply notify the `ball` server that it was maintaining a cache of b's object attributes and cache the color of the ball in a local state variable. Then when the `read_color` operation was requested, the local proxy object could reply `red` without any further communication with the server.

Proxy objects raise many of the same issues that memory objects do. They must define protocols for consistency and caching that are appropriate for the objects they represent. However, we can resolve these issues by applying the same techniques that we used to implement shared memory objects. Furthermore, because proxy objects are specifically generated to cooperate with their servers, we can instantiate arbitrarily complex protocols for consistency and caching without burdening the user with this complexity. We believe that proper use of proxy objects substantially increases the flexibility of Photon language veneers and opens up new opportunities to improve the performance of distributed applications in high-latency environments.

### 5.10.4. Description of C++ Implementation of Futures.

A future is an object that will return a value when claimed by a program. The value is typed -- in Photon, the future itself may be statically typed according to the type of the value it will return, or may be untyped (i.e. dynamically typed.) The difference between a variable of a type and a future of that type is that the future may or may not contain a value at any particular time, and contains state information indicating whether it does or does not contain the value. A variable of a type always contains a value, even if it is uninitialized, and it would not make sense to use this value.

When a program claims the value of the future, it will receive the value immediately if the value is already available, or it it will wait until the value is available. This must make sense for the program to be useful—some other active part of the program, or some other program, must be prepared to supply the value. The future is clearly an inter-process communication facility. In Photon, we provide facilities that allow a program to create a future that may then be used to pass a value between two other programs which may be on different machines. This allows a program to direct a distributed computation without being an intermediary for data exchange. It also eliminates needless delays for messages passing through the network that would occur if the program was the intermediary—improving the performance of the distributed computation.

Example:

In C++, a Future is an object, declared as `Future<type>`, a template defined by the Photon definition files. The value type of a future may be any data type. A declaration like the following:

```
Future<int> f;
```

declares an object `f` which is an `int`-returning future. This object may be *supplied* by use of an assignment statement:

```
f = 5;
```

or *claimed* by something like:

```
int i; i = f;       // or by "if(f)", etc.
```

If a program containing the first declaration of `f` above executes the first statement in one thread and the second in a different thread, the second thread will wait until the value is available from the first thread and then receive the value supplied by the other thread. In Photon, these threads may be in the same program or in different programs, and on the same machine or in different machines.

An untyped future may also be declared as follows:

```
Future untyp;
```

An untyped future may be supplied using `Supply(type)` or claimed using `Claim(type)`. For example:

```
integer i;
untyp.Supply(i);
untyp.Claim(i);     // useful if in some
                    // other thread
```

When the `Claim` method is executed, the thread executing the `Claim` will wait until a `Supply` is executed on the same future. It will then receive this value, provided that its argument is of sufficient size and of compatible type with the value dynamically bound to the future.

Untyped futures are particularly useful for transit futures when no use is made value of the future, or of its type in the invoking program.

### 5.10.5. Photon Objects and Methods.

Since Photon is object-oriented, one usually refers to an object, not directly to a program. Of course, whenever a method is executed on some particular machine, there must be a process or thread to execute that method. For the duration of its execution, a method invocation is equivalent to a thread. The following ODL statements:

```
type Random

   abbrev is Rand

   representation is state: float;

   operation get()
   returns (result: float);

 end type Random;

 type Accumulator

   abbrev is Accum
   representation is value: float;

   operation add(input: float)
   returns (output: float);

 end type Accumulator;
```

defines two Photon classes, `Random` and `Accumulator`. If the get method is invoked on a `Random` object, a float of random value is returned.

This definition is translated into the following C++ class definition.

```
class Random
  {
    private:
   float state;
    public:
   float get();
  };

 class Accumulator
  {
    private:
   float value;
    public:
   float add(float);
  };
```

These statements would be obtained by running the Photon class tool with suitable arguments, and storing the output in one or more files named after the classes they define. These normally would then be included by programs that make uses of these Photon classes.

In C++, the following program:

```
#include "Random.H"
#include "Accumulator.H"

main()
{
Random r("hostA");
Accumulator a("hostB");
float f = r.get();
a.add(f);
}
```

first locates a Random object, initializing a proxy object r to refer to it, then locates an Accumulator object, calling its proxy a. It then invokes the get method on r, receiving a float value which it stores in variable f. Finally, this value is added to object a's total by invoking the Accumulator class method add.

### 5.10.6. Setup Facilities

An important use of Futures in Photon results from the ability to use a future to specify a flow of data between the methods of two objects. Any formal parameter of a method may have an actual which is either of the type it specifies, or a future of that type. When the same future is specified as an output parameter to one method invocation and as an input parameter to another method invocation, Photon arranges that the value that the first method supplies to the future (as the output parameter) is supplied as the input parameter to the other method invocation. Continuing to use the Random and Accumulator classes from the previous example, we can construct the following program:

```
main() {
    Random r("hostA");
    Accumulator a("hostB");
    Future<int> f;
    f = r.get();
    a.add(f); }
```

This program does the same thing as the one in the previous example, but variable f is a Future<int>—i.e. a future returning an int. In this program, the value of the future is neither produced or consumed by main, and so, it does not need to pass through the local computer. Instead, the value passes directly from hostA to hostB.

### 5.10.7. Example of Future Parameters

In the Photon Object Definition Language, a parameter of any method may be declared to be of any declared type in the abstract type space defined by Photon, or a Future of that type. A future is not itself a Photon abstract data type, but is a type defined by the language support for C++. The method function declaration generated by the ODL processor will contain Future<type> in the appropriate parameter positions if "Future type " was declared in the ODL definition of the method. Photon takes appropriate action to handle each of the 8 distinct cases resulting from the combination of input or output parameter, formal future or non-future parameter, actual future or non-future parameter.

When a formal parameter is not a future, the value is transported either before or after execution of the method, depending on whether it is an input or output parameter. When the formal of an input parameter is a future, execution of a method is initiated immediately, whether the value of the future is available or not. The method will block when (or if) it tries to claim this value. When the formal of an output parameter is a future, the value is transported to its destination as soon as it becomes available during the execution of the method.

# 6. Issues revisited

The previous section described the key technologies being incorporated in Photon to provide high performance in what we believe to be the computational environment of the 1990s and beyond. In this section, we revisit the issues we raised as the result of our sample application analysis and examine how Photon has addressed them. Those areas that have not been addressed have been noted as areas for further research.

## 6.1. Communications

*How do we deal with increased communications latency?* We have incorporated sequences and global futures within Photon to accommodate increased communications latency. By using futures to pipeline requests and replies, we can cause many operations to one server to be executed in only one round-trip communications time. We can further use futures to cause data to flow directly from the producer of data to consumers of that data, thereby further reducing latency effects.

*How do we model scheduled versus unscheduled communications (e.g., connection-based models versus datagram models)?* Photon can use both connection-oriented and non-connection-oriented communications models for its communications substrate. Because of its layered architecture, most Photon mechanisms operate independently of the underlying physical communications layers. Where special communications features are required by applications, objects can be bound selectively according to specific communication parameters such as reliable transport or fixed latency. The application has this flexibility because of Photon's explicit location of objects.

*What communications semantics do we guarantee (e.g., at-most-once delivery versus at-least-once delivery or best effort delivery)?* Photon typically provides sequenced, reliable data streams as its normal, high-bandwidth communications mechanism. However, applications can specify their communications needs at the time of object location to permit the use of less expensive communications methods. Furthermore, proxy objects permit partitioning of object methods to guarantee application-specific consistency requirements regardless of the intervening communications layer.

*How does the user specify the communication needs of the application?* The user specifies the communications needs of the application through additional parameters passed to the *locate* operation. These parameters tell the system what aspects (e.g., cost, bandwidth, latency) of the communications path to optimize.

*How do we provide communications to many objects at once?* Logical Object identifiers (LOIDs) provide a mechanism for referring to multiple instances of objects at once. We have designed this facility largely to permit replication of objects for fault-tolerance. However, LOIDs can also be resolved into multicast physical identifiers where the underlying communication substrate permits this type of communication This mapping permits the addressing of multiple physical objects in one operation. This use of multicasting is most appropriate when the application is performing one-way communication or when custom proxy objects are being used that can coordinate responses from multiple objects; otherwise, there is the opportunity to confuse the application with multiple replies to one request.

*How do communications abstractions interact with the object-oriented programming model?* The programmer at the object level in Photon interacts with the communications abstraction largely through the use of parametrized location of objects and through the use of global futures. Objects can be located by attribute through the use of a Photon location broker. Global futures then permit the application to specify the data flow between objects independently of the control flow of the program. Both of these mechanisms can be encapsulated cleanly into an object's definition through the use of language veneers.

## 6.2. Computation

*How do we express parallelism in a distributed application?* Logical parallelism is expressed in the application through the use of the future abstraction. Futures allow the programmer to define parallelism within the application while simultaneously establishing a point of synchronization for the results of that parallelism. Futures allow the programmer to express opportunities for parallelism without necessarily requiring parallel resources to be allocated.

*How do we bind our parallelism abstractions to real computational elements to maximize the performance of the computation?* While futures define opportunities for parallelism, it is up to the resource management policies of the system to define what actual resources will be used to achieve this parallelism. Photon does not attempt to restrict the policies that might be used to accomplish this resource allocation. These policies are defined by how the objects being invoked are implemented by servers and how the application locates these objects. Futures permit us to understand how much inter-object parallelism the application can use and still execute correctly. Nonetheless, even more parallelism may be used within an object method provided that it does not violate any of its ODL specifications; therefore, futures simply provide us with a lower bound on the amount of parallelism that might be used.

*How do our computational abstractions synchronize with one another?* All object invocations return a result that can be used for synchronization. Synchronous operations are accomplished by having the application immediately claim this synchronization result. Asynchronous operations allow the client to continue executing until the result of the operation is needed. Futures involve storing or passing this synchronization result to other clients or servers.

### 6.3. Binding

*How do we create bindings between clients and servers?* Clients can create bindings to servers by locating physical object identifiers (POIDs) for objects they wish to perform operations on. These POIDs contain all information required to communicate with the object. POIDs are usually obtained from an object location broker or name server.

*At what time are these bindings created?* A client is bound to a server when a *locate* operation is performed on a logical object ID (LOID) or when an initial operation is done on a POID. Both of these functions are usually performed when the object name is resolved and the object is *imported* into the application.

*How long do these bindings last?* Bindings are either explicitly destroyed when an application terminates or are timed out after a period of inactivity. In either case, a new locate operation will have to be performed before any new operations can be sent from that client to that server.

### 6.4. Naming

*How do we name objects and services?* Services and objects can have three types of names: symbolic names, logical names (LOIDs) , and physical names (POIDs).

*How do we map descriptive requirements (names, properties, keywords) to servers?* Just as symbolic names can be mapped to logical names by a name server, descriptive requirements can be similarly mapped through use of a location broker. In both cases, a logical name for a service is returned by a lookup in a database; the name server database is indexed by symbolic name, and the property server database is indexed by attribute.

*What format and scope does an object name have?* Symbolic names are alphanumeric strings, and LOIDs and POIDs are fixed-length structures. All names have system-wide scope.

*Can an object name specify more than one object?* A symbolic name can specify only one object, a logical or physical name. A logical name can specify many physical names that it can resolve to. A physical name by definition specifies only one object

### 6.5. Programming Model Issues

*What is the model of computation offered to the programmer?* The fundamental model of computation used in Photon is an object-oriented model. In this model, the fundamental addressable atom throughout the system is the object. All operations upon objects are performed by methods whose interfaces are publicly defined. These methods are gathered together into servers that act as computational representatives for internally stored objects. In some implementations, portions of methods may be incorporated into a client application by means of proxy objects.

*How is this computation model integrated into the programming language?* The object-oriented model implemented by Photon is integrated into programming languages through language veneers. The language veneer is a set of support routines, definitions, macros, and preprocessors (if necessary) to support Photon concepts within that language in the most natural form possible. In the case of extensible languages such as C++, this veneer can be done almost entirely through libraries and class definitions.

### 6.6. Fault tolerance and Correctness

*How do we provide fault tolerance in a high-latency communications environment?* This is an area for further research. Photon supplies basic mechanisms for replicated objects through its use of proxies and logical object addresses. However, it does not yet define replication techniques or policies that attempt to provide general fault-tolerance. We have speculated that in high-latency communications environments, primary copy schemes will often be more efficient than arbitrary replication protocols. This efficiency occurs because locking and synchronization among records can be done entirely within the primary copy without reference to any other servers. While updates must still be propagated to secondary servers before they can be considered complete, the reduced need for synchronization decreases the number of round-trips required between copies of the server, and therefore will provide higher-performance operation.

In the case of failure of the primary copy, a new primary is elected by the remaining servers for that service. Because this voting process need only be done on failure of a primary server (rather than on the failure of any server in quorum schemes or on every operation in version voting), primary copy schemes result in fewer changes in service due to failure.

*Can we provide apparent atomicity of complex operations involving many objects?* We have not yet addressed how to achieve atomicity of execution for operations involving many objects. Transactions provide a powerful means of guaranteeing atomicity of such operations, but transactions also impose significant costs on the programs using them. For example, every operation that can be part of a transaction must have a way of rolling back its state at any point until a `commit` is done. Furthermore, transactions require multiple round-trips over the communications path to specify both the operation invocations and the commits required to post them. We will continue to look into ways in which we can provide atomicity of multiple operations with high performance and low cost.

## 6.7. Resource Allocation

*How do we optimize the use of resources for a computation? How do we determine the correct dimension of the application to optimize (e.g., performance, security, correctness)?* Our architecture does not presently attempt to dictate how resources are optimized. Instead, Photon provides facilities such as dynamic method binding and proxy objects that allow processes to control their resource use according to its needs. These mechanisms can be be manipulated by interactive processes to achieve performance optimization on a local scale. However, global optimization of resource use and general-purpose implementation of resource management policies are beyond the scope of this project and are best addressed through the use of cooperating Photon servers, each providing local resource management. Photon will provide interfaces for administrators to attach servers that implement desired resource management policy.

# 7. Comparisons with Related Work

Photon incorporates concepts from a wide variety of other systems. In this section, we compare and contrast the mechanisms in Photon with those provided by other state-of-the-art systems.

## 7.1. Cronus

Cronus is a distributed programming environment designed to work on a wide v riety of heterogeneous operating systems and platforms [Schantz Cronus][Vinter Cronus]. It was developed at BBN under funding from the Rome Laboratory of the United States Air Force.

Cronus provided the model for many of the mechanisms in the object layer. We were particularly influenced by Cronus's ability to provide an object-oriented programming environment on top of a wide variety of non-object-oriented operating systems and computer architectures. Photon incorporates several specific Cronus concepts such as location-independent object identifiers, futures, and direct connections ith little or no change.

The layering model in Photon was inspired by the Cronus philosophy of running on top of a native operating system. Photon extended that philosophy to incorporate the concept of a process-to-instance mapping for resource management. Nonetheless, an existing native operating system will make an excellent instance layer for Photon and should substantially decrease the cost of implementation.

Our principal differences with Cronus come from our interest in being able to bind clients to servers for the purpose of high-performance communications. Cronus uses a purely location-independent model of addressing objects, where Photon uses both location-independent and location-dependent names for objects. Photon's ability to associate an object with a communications path permits the use of streaming for communication, thereby increasing communications performance and decreasing the effects of latency. Cronus assumes that all operations on an object are independent of one another, and this makes it difficult to do streaming without considerably more mechanism.

Cronus has a well-defined quorum-voting scheme for achieving fault tolerance by means of replicated servers. While Photon could implement a similar facility, we believe that the synchronization costs of replicated servers will be much high than those inherent in primary copy schemes. Therefore, Photon's fault-tolerance architecture will rely less on version-voting methods. Nonetheless, we have found the Cronus concept of automatically generating fault-tolerant servers according to a specification to be quite important to users, and we will provide similar techniques in our system.

## 7.2. Mercury and Argus

Mercury is a distributed operating system developed at MIT [Liskov Mercury]. Argus is both a programming language and a distributed system built on top of Mercury primitives [Liskov Argus]. Both systems were developed under funding from the Defense Advanced Research Projects Agency and the National Science Foundation.

The Mercury and Argus systems contributed the concepts of call streams and data flow communications using RPC constructs to the Photon architecture. Argus also supports promises (essentially strongly typed futures used for streaming and thread synchronization), the use of transactions for atomicity, language veneers, and clearly defined exception-handling semantics.

Argus has significant capabilities in the area of fault tolerance. In particular, it supports *atomic transactions* or *actions*. Actions are simply operations that are guaranteed to complete atomically or to not happen at all, regardless of failures. Therefore, Argus programs are able to guarantee the correctness of the data they control despite arbitrary failures in the distributed system. This capability in conjunction with primary-copy replicated services results in extremely good tolerance for system and component failure.

Attractive though it is, we have not adopted atomic transactions for Photon, largely due to uncertainty regarding their cost in high-latency environments. On one hand, transactions allow the buffering of many operations into one entity, which should improve communications throughput. However, the need for two-phase protocols to commit operations imposes a need for at least two round-trips in addition to those required for the original operations. Therefore, we have left the use of transactions in Photon as an open question for further consideration.

## 7.3. Alpha

Alpha is an adaptable decentralized operating system for real-time applications [Jensen Alpha]. It was initially developed at Carnegie-Mellon University, and later versions are being developed at Concurrent Computer Corporation under funding from the Rome Laboratory of the United States Air Force.

Alpha is specifically optimized for integration and optimization of large, complex, distributed real-time systems. Alpha provides the concept of threads that are scheduled according to time-value functions that dictate their priority over time. These threads can migrate from host to host in a distributed system and carry with them their scheduling policies, access rights, and resource management requirements. Alpha, like Argus, provides transactions to ensure atomicity of function; however, Alpha's transaction mechanisms are integrated with the scheduling policies of the system to guarantee satisfaction of real-time constraints.

Alpha's decoupling of scheduling policy from mechanism is similar in spirit to Photon's decoupling of the communications mechanisms from the programming model. Our concept in Photon is that the programmer is free to construct processes and abstract objects without regard for where or how they will be instantiated at run time. When the actual application is run, the instantiation and location of those objects and processes is dictated by user-defined resource management policies enforced by the system. Photon extends this concept by optimizing the communications path also at this time through dynamic method binding.

While Photon is optimized for high performance, it is not specifically targeted at solving real-time distributed applications as Alpha is. Therefore, the tradeoffs made in Photon are directed more toward flexibility, scalability, and ease of use than toward real-time needs. We believe this emphasis is justified due to the increasing ability of organizations to dedicate hardware to solving real-time problems and the continuing requirement for systems that are easy to use and understand.

## 7.4. Amoeba

Amoeba is a distributed system developed at the Vrije Universiteit (VU) in Amsterdam, the Netherlands by Prof. Andrew Tanenbaum [Tanenbaum Amoeba].

Amoeba is particularly interesting for this project because it is a distributed system designed specifically to work with parallel processors. Amoeba provides a client/server architecture using a microkernel. Processors are considered to be well-connected nodes in the distributed system and are allocated dynamically to users. Amoeba's communication is based upon a very-high-performance remote procedure call. Objects are referenced by means of *capabilities*, which are cryptographically protected, location-dependent names. Naming of objects is provided through a directory server.

Photon's use of location-dependent names for high performance is similar to that of Amoeba's use of capabilities. However, Photon provides location-independent names also to permit transparent replication of services. Amoeba's capabilities also dictate access rights to a server, whereas Photon will use authentication, tickets, and access control lists such as in Cronus to exercise access control.

Amoeba achieves much of its high communication performance through the use of tightly coded kernel routines and minimal services. The Amoeba kernel runs directly on the hardware and does not provide features such as paging or swapping. Photon takes a somewhat different tack because it is designed to accommodate a resident operating system at the instance layer. Therefore, Photon provides high-level mechanisms such as global futures to achieve high throughput and uses implementation techniques such as dynamic method binding to reduce layering costs and data copying.

# 8. Photon Demonstrations

During this project, we developed a series of demonstration programs to illustrate the fundamental mechanisms that make up Photon. The Photon Concept Demo software is designed to illustrate some of the basic Photon features in simplified form. The demonstration software uses a kernel-less, client-server model, in which the programs making up each demonstration are run as unprivileged, user processes. The Photon functionality needed is built into a library that is linked into each program. This software is supplied in both source and binary form for the Sun 3 and Sun 4 platforms. This same demonstration software has also been compiled and run on Apollo DN10000 parallel processors.

The demonstration software shows some of the mechanisms critical to high performance in Photon. These include the following concepts:

- synchronous and asynchronous Remote Procedure Calls (RPCs)
- stream and non-stream RPCs, and global futures. These capabilities are built on top of the native operating system's TCP socket interface
- demonstratable versions of our globally-claimable futures mechanism that show third-party transfers, pipelining, and sequences
- a prototype Photon support library in C
- a simple name server to locate Photon services
- a simple prototype of the C++ language veneer.

These concepts are encapsulated into clients and servers that can be used to demonstrate the following functions:

- Name server (name_server, phls)
- Synchronous Echoing (echo_client_1 -de 1, echo_server)
- Pipelined Echoing (echo_client_1 -de 5, echo_server)
- Third-party Data Transfer (echo_client_2)
- Sequences (seq_send, seq_recv)
- Mandelbrot Parallel Computation (mandel_client, mandel_server)
- Echoing using a C++ Photon veneer (Echo2, Echo2b, Echo3, ftyped)

For complete details regarding the demonstration software, please refer to the Photon Software User's Manual, BBN Report #7709.

# 9. Areas for Further Research

In this project, we developed a collection of key concepts that form the foundations for a latency-independent, high performance distributed environment. These concepts were:

- Object architectures
- Global futures
- Sequences
- Distributed shared memory
- Explicit location mechanisms
- Language veneers

While these concepts form a consistent architecture for achieving the goals of this project, they need further examination and extension in the context of requirements for fault-tolerance and resource management. We believe that work in the following areas would be beneficial:

- More complete designs of the core communication elements, such as futures, sequences, and shared memory objects.
- Process abstractions including threads and scheduling
- Designs of key services, such as location broker, name server, and type management, to support reconfiguration, resource management, symbolic naming, and application development. These services must be designed to augment Photon, while allowing continued operation of Photon communication when they fail.
- More attention to reliability, including reliable, at-most-once delivery of requests and object replication
- Approaches for allocating, sharing, and restricting use of resources such as processor time and memory, communication bandwidth and transport connections

We believe the last two points are particularly important in $C^2$ environments. For Photon to develop into a survivable and robust system, we need to develop techniques for error detection and recovery, and we need to prove that these structures are sufficient to support system operation in the presence of network partitions and component failures. We believe that modern techniques for software fault tolerance, distributed database integrity, reliable transaction transport, and multiprocessor shared memory synchronization would all be appropriate candidates for enhancing the robustness of Photon.

In the area of resource management, Photon currently does not define how servers are propagated to new hosts. We could define a new element of the architecture to locate code for a particular application component, to transmit the code to an idle computational node, and to start it running. Combining this with an approach to grouping nodes and applications, and assigning a class of applications to a class of nodes would provide a focus for designing segmented processor resource allocation strategies.

Finally, we believe that interoperation with other distributed systems and languages will become increasingly important as our research proceeds. Large investments have already been made in distributed systems such as Cronus and Alpha. As more organizations deploy local and wide area networks to meet their communications needs, more applications will become available for integration into our distributed environment. If we can preserve the investments that have been made in previous distributed systems while integrating the concepts we have developed in Photon, we should then be able to substantially reduce the development time for new applications of distributed system technology. Reduced development time permits greater responsiveness to users and can reduce overall system costs.

## 10. Summary

In this paper, we have described the issues and solution mechanisms involved in the design of a high-performance distributed system architecture. We have examined both the characteristics of next-generation communications systems and the requirements of simple $C^2$ distributed applications. We have designed a basic architecture and mechanisms for a distributed system called Photon that is capable of meeting the needs of these applications in high-performance environments. While this is only the first step in the development of high performance distributed systems, we believe that the concepts described here can make substantial contributions to next-generation architectures. By performing this work now, we have prepared the way for software architectures capable of exploiting the vast potential in fiber-optic communications and scalable distributed computation. We believe that systems like these will provide the technology to serve the needs of $C^2$ users well into the next decade and beyond.

# References

[Apollo NCS/NCA]
Zahn, Lisa, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Waynt, *Network Computing Architecture*. Prentice Hall, Englewood Cliffs, NJ 07632, 1990.

[Bennett Munin]
Bennett, J., Carter, J., and Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", *Proc. 1990 Conference on Principles and Practice of Parallel Programming*, ACM Press, New York, N.Y., 1990, pp. 168-176.

[Berets Cronus]
Berets, James C. and Richard M. Sands, "Introduction to Cronus," BBN Systems and Technologies Corporation, Technical Report 6986, January 1989.

[Cronus Multicluster]
"Integrated Distributed Computing for Wide-area Network Environments," BBN Proposal P89-LABS-C-088, BBN Systems and Technologies Corporation, November 1988.

[Dasgupta Clouds]
Dasgupta, P. and M. Morsi, "An Object-Based Distributed Database System Supported on the Clouds Operating System." GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[EE Times]
Electronic Engineering Times, Issue 677, January 27, 1992, p 4.

[Estria IDRP]
Estrin, D., Y. Rekhter, and S. Hotz, "A Unified Approach to Inter-Domain Routing," Internet Draft Working Paper.

[Halstead Multilisp]
Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, October 1985.

[Jensen Alpha]
Northcutt, J. Duane and E. Douglas Jensen, Raymond K. Clark, James G. Hanko, Donald C. Lindsay, David P. Maynard, Franklin D. Reynolds, Samual E. Shipman, Jack A. Test, and Jeffrey E. Trull, "Decentralized Computing Technology for Fault-Tolerant, Survivable C3I Systems", Final Technical Report, November 1988.

[Joseph ISIS]
Joseph, Thomas A., "Exploiting Virtual Synchrony in Distributed Systems," TR 87-811, Department of Computer Science, Cornell University, Ithaca, New York 14853-7501, February 1987.

[LeBlanc Clouds]
LeBlanc, Richard J. and C. Thomas Wilkes, Systems Programming with Objects and Actions, pp. 1-20, March, 1985. Georgia Institute of Technology Technical Report GIT-ICS-85/03

[Liskov CLU]
Liskov, Barbara, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, Springer-Verlag, 1984.

[Liskov Argus]
Liskov, Barbara, "The Argus Language and Systems," Lecture Notes in Computer Science 190, pp. 343-430, Springer-Verlag, 1985.

[Liskov Mercury]
Liskov, Barbara, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl, "Communication in the Mercury System," *Proc. of the Twenty First Annual Hawaii International Conference on System Sciences*, pp. 178-187, January 1988.

[Ladin Availability]
Ladin, Rivka, B. Liskov, L. Shira, "A Technique for Constructing Highly-Available Services, MIT Laboratory for Computer Science, Technical Report MIT/LCS/TR-409.

[Microprocessor Report]
Microprocessor Report, October 1989, published by Microdesign Resources.

[Northcutt Alpha]
Northcutt, J. Duane, *Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel*, Academic Press, Inc., 1987.

[Rashid MACH]
Rashid, R. F., "Threads of a New System," *Unix Review*, Vol. 4, No. 8, Aug. 1986, pp. 37-49.

[Schantz Cronus]
Schantz, Richard E. and Robert H. Thomas, "Cronus, A Distributed Operating System: Functional Definition and System Concept," BBN Laboratories, Technical Report 5879, June 1982, Revised January 1985.

[Schroder Cronus TVE1]
Schroder, Kenneth J., "Cronus TVE Functional Description," BBN Systems and Technologies Corporation, Technical Report 6838, July 1987, Revised December 1988.

[Schroder Cronus TVE2]
Schroder, Kenneth J., Jonathan G. Cole, and Chantal Eide, "Cronus TVE Final Technical Report," BBN Systems and Technologies Corporation, Technical Report 6841, January 1989.

[Steenstrup IDPR]
Lepp, M. and M. Steenstrup, "An Architecture for Inter-Domain Policy Routing," Internet Draft Working Paper, July 1991.

[Steenstrup UK FatPipe]
Steenstrup, Martha, "A Rate-Based Congestion Control Algorithm for an Internetwork," Draft working paper, unpublished.

[Steiner Kerberos]
Steiner, Jennifer G., Clifford Newman, and Jeffrey I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *Proceedings of the Winter USENIX 1988 Conference*.

[Sun RPC]
"RPC: Remote Procedure Call Protocol Specification," Internet Request for Comments 1050, April 1988.

[Tanenbaum Amoeba1]
Mullender, S. J. and A. S. Tanenbaum, "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5-6, pp. 421-432, October 1984.

[Tanenbaum Amoeba2]
Mullender, Sape J., Guido van Rossum, Andrew S. Tanenbaum, Robert van Renesse, and Hans van Staveren, "Amoeba--A Distributed Operating System for the 1990s".

[Tanenbaum Surveys]
Tanenbaum, A. S., and Van Renesse, R. "Distributed Operating Systems", *Computing Surveys 17* (Dec. 1985) 419-470.

[Tokuda Real-Time Mach]
Tokuda, Hideyuki, "Real-time Mach," Presentation to NGCR Meeting, January 25, 1990.

[Walker Cronus futures]
Walker, Edward F. and Richard Floyd, "Asynchronous Remote Operation Execution in Distributed Systems," *Proceedings of the Tenth International Conference on Distributed Computing Systems*, pp. 253-259, IEEE Computer Society, May 1990.

[X3S3.3 IDRP]
"ASC X3S3.3. Intermediate System to Intermediate System Inter-Domain Routing Exchange Protocol," Working Document 90-216, ANSI, New York, July 1990.
[Vinter Cronus]
Vinter, S., "Integrated Distributed Computing Using Heterogeneous Systems," *IEEE Signal*, June 1989.

[Zhang VirtualClock]
Zhang, Lixia, "VirtualClock: a new traffic control algorithm for packet-switched networks," *ACM Transactions on Computer Systems*, vol. 9, no. 2, pp. 101-124, May 1991.

*U.S. GOVERNMENT PRINTING OFFICE